Джек Креншоу
Давайте создадим компилятор!

Аннотация

Эта серия, написанная в период с 1988 по 1995 года и состоящая из шестнадцати частей, является нетехническим введением в конструирование компиляторов. Серия является руководством по теории и практике разработки синтаксических анализаторов и компиляторов языков программирования. До того как вы закончите чтение этой книги, вы раскроете каждый аспект конструирования компиляторов, разработаете новый язык программирования и создадите работающий компилятор.

Давайте создадим компилятор!

Введение

ВВЕДЕНИЕ

Эта серия статей является руководством по теории и практике разработки синтаксических анализаторов и компиляторов языков программирования. Прежде чем вы закончите чтение этой книги, мы раскроем все аспекты конструирования компиляторов, создадим новый язык программирования, и построим работающий компилятор.

Хотя я по образованию и не специалист в компьютерах, я интересовался компиляторами в течение многих лет. Я покупал и старался разобраться с содержимым практически каждой выпущенной на эту тему книги. И, должен признаться, это был долгий путь. Эти книги написаны для специалистов в компьютерной науке и слишком трудны для понимания большинству из нас. Но с течением лет часть из прочитанного начала доходить до меня. Закрепить полученное позволило то, что я начал самостоятельно пробовать это на своем собственном компьютере. Сейчас я хочу поделиться с вами своими знаниями. После прочтения этой книги вы не станете ни специалистом, ни узнаете всех секретов теории конструирования компиляторов. Я намеренно полностью игнорирую большинство теоретических аспектов этой темы. Вы изучите только практические аспекты, необходимые для создания работающей системы.

В течение всей книги я буду проводить эксперименты на компьютере, а вы будете повторять их за мной и ставить свои собственные эксперименты. Я буду использовать Turbo Pascal 4.0 и периодически буду включать примеры, написанные в TP. Эти примеры вы будете копировать себе в компьютер и выполнять. Если у вас не установлен Turbo Pascal вам будет трудно следить за ходом обучения, поэтому я настоятельно рекомендую его поставить. Кроме того, это просто замечательный продукт и для множества других задач!

Некоторые тексты программ будут показаны как примеры или как законченные продукты, которые вы можете копировать без необходимости понимания принципов их работы. Но я надеюсь сделать гораздо больше: я хочу научить вас КАК это делается, чтобы вы могли делать это самостоятельно, и не только повторять то что я делаю но и улучшать.

Такую задачу не решить на одной странице. Я попытаюсь сделать это в нескольких статьях. Каждая статья раскрывает один аспект теории создания компиляторов и может быть изучена в отдельности от всех других. Если вас в настоящее время интересует только какой-то определенный аспект, тогда вы можете обратиться к нужной статье. Каждая статья будет появляться по мере завершения, так что вы должны дождаться последней для того, чтобы

считать себя закончившими обучение. Пожалуйста, будьте терпеливы.

В общем, каждая книга по теории создания компиляторов раскрывает множество основ, которые мы не будем рассматривать. Типичная последовательность:

• вступление, в котором описывается что такое компилятор.

• одна или две главы, описывающие задание синтаксиса с использованием формы Бэкуса-Наура (БНФ).

• одна или две главы с описанием лексического анализа, с акцентом на детерминированных и недетерминированных конечных автоматах.

• несколько глав по теории синтаксического анализа, начиная с рекурсивного спуска и заканчивая LALR анализаторами.

• глава, посвященная промежуточным языкам, с акцентом на P-код и обратную польскую запись.

• множество глав об альтернативных путях для поддержки подпрограмм и передачи параметров, описания типов, и т.д.

• завершающая глава по генерации кода, обычно для какого-нибудь воображаемого процессора с простым набором команд.

• финальная глава или две, посвященные оптимизации. Эта глава часто остается непрочитанной, очень часто.

В этой серии я буду использовать совсем другой подход. Для начала, я не остановлюсь долго на выборе. Я покажу вам путь, который работает. Если же вы хотите изучить возможности, хорошо… я поддержу вас... но я буду держаться того, что я знаю. Я также пропущу большинство тех теорий, которые заставляют людей засыпать. Не поймите меня неправильно: я не преуменьшаю важность теоретических знаний, они жизненно необходимы, когда приходится иметь дело с более сложными элементами какого либо языка. Но необходимо более важные вещи ставить на первое место. Мы же будем иметь дело с методами, 95% которых не требуют много теории для работы.

Я также буду рассматривать только один метод синтаксического анализа: рекурсивный спуск, который является единственным полностью пригодным методом при ручном написании компилятора. Другие методы полезны только в том случае, если у вас есть инструменты типа Yacc, и вам совсем неважно, сколько памяти будет использовать готовый продукт.

Я также возьму страницу из работы Рона Кейна, автора Small C. Поскольку почти все другие авторы компиляторов исторически использовали промежуточный язык подобно P-коду и разделяли компилятор на две части («front end», который производит P-код, и «back end», который обрабатывает P-код, для получения выполняемого объектного кода), Рон показал нам, что очень просто заставить компилятор непосредственно производить выполняемый объектный код в форме языковых утверждений ассемблера. Такой код не самый компактный в мире код... генерация оптимизированного кода – гораздо более трудная работа. Но этот метод работает и работает достаточно хорошо. И чтобы не оставить вас с мыслью, что наш конечный продукт не будет представлять никакой ценности, я собираюсь показать вам как создать компилятор с небольшой оптимизацией.

Наконец, я собираюсь использовать некоторые приемы, которые мне показались наиболее полезными для того, чтобы понимать, что происходит, не продираясь сквозь дремучий лес. Основным из них является использование односимвольных токенов, не содержащих пробелов, на ранней стадии разработки. Я считаю, что если я могу создать синтаксический анализатор для распознавания и обработки I-T-L, то я смогу сделать тоже и с IF-THEN-ELSE. На втором уроке я покажу вам, как легко расширить простой синтаксический анализатор для поддержки токенов произвольной длины. Следующий прием состоит в том что я полностью игнорирую файловый ввод/вывод, показывая этим что если я могу считывать данные с клавиатуры и выводить результат на экран я могу также делать это и с файлами на диске. Опыт показывает, что как только транслятор заработает правильно очень просто перенаправить ввод/вывод на файлы. Последний прием заключается в том, что

я не пытаюсь выполнять коррекцию/восстановление после ошибок. Программа, которую мы будем создавать, будет распознавать ошибки и просто остановится на первой из них, точно также как это происходит в Turbo Pascal. Будут и некоторые другие приемы, которые вы увидите по ходу дела. Большинство из них вы не найдете в каком либо учебнике по компиляторам, но они работают.

Несколько слов о стиле программирования и эффективности. Как вы увидите, я стараюсь писать программы в виде маленьких, легко понятных фрагментов. Ни одна из процедур, с которыми мы будем работать, не будет состоять из более чем 15-20 строк. Я горячий приверженец принципа KISS (Keep It Simple, Sidney – Делай это проще, Сидней) в программировании. Я никогда не пытаюсь сделать что-либо сложное, когда можно сделать просто. Неэффективно? Возможно, но вам понравится результат. Как сказал Брайан Керниган, сначала заставьте программу работать, затем заставьте программу работать быстро. Если позднее вы захотите вернуться и подправить что-либо в вашем продукте, вы сможете сделать это т.к. код будет совершенно понятным. Если вы поступаете так, я, тем не менее, убеждаю вас подождать пока программа не будет выполнять все, что вы от нее хотите.

Я также имею тенденцию не торопиться с созданием модулей до тех пор, пока не обнаружу, что они действительно нужны мне. Попытка предусмотреть все необходимое в будущем может свести вас с ума. В наши время, время экранных редакторов и быстрых компиляторов я буду менять модули тогда, когда почувствую необходимость в более мощном. До тех пор я буду писать только то, что мне нужно.

Заключительный аспект: Один из принципов, который мы будем применять здесь, заключается в том, что мы не будем никого вводить в заблуждение с P-кодом или воображаемыми ЦПУ, но мы начнем с получения работающего, выполнимого объектного кода, по крайней мере, в виде программы на ассемблере. Тем не менее, вам может не понравиться выбранный мной ассемблер… это – ассемблер для микропроцессора 68000, используемый в моей системе (под SK*DOS). Я думаю, что вы найдете, тем не менее, что трансляция для любого другого ЦПУ, например 80x86, совершенно очевидна, так что я не вижу здесь проблемы. Фактически, я надеюсь что кто-то, кто знает язык 8086 лучше, чем я, предоставит нам эквивалент объектного кода.

ОСНОВА

Каждая программа нуждается в некоторых шаблонах … подпрограммы ввода/вывода, подпрограммы сообщений об ошибках и т.д. Программы, которые мы будем разрабатывать, не составляют исключения. Я попытался выполнить их на минимальном уровне, чтобы мы могли сконцентрироваться на более важных вещах и не заблудиться. Код, размещенный ниже, представляет собой минимум, необходимый нам, чтобы что-нибудь сделать. Он состоит из нескольких подпрограмм ввода/вывод, подпрограммы обработки ошибок и скелета – пустой основной программы. Назовем ее Cradle. По мере создания других подпрограмм, мы будем добавлять их к Cradle и добавлять вызовы этих подпрограмм. Скопируйте Cradle и сохраните его, потому что мы будем использовать его неоднократно.

Существует множество различных путей для организации процесса сканирования в синтаксическом анализаторе. В Unix системах авторы обычно используют getc и ungetc. Удачный метод, примененный мной, заключается в использовании одиночного, глобального упреждающего символа. Части процедуры инициализации служит для «запуска помпы», считывая первый символ из входного потока. Никаких других специальных методов не требуется… каждый удачный вызов GetChar считывает следующий символ из потока.

```
{--------------------------------------------------------------}
program Cradle;
{--------------------------------------------------------------}
```

```
 Constant Declarations
const TAB = ^I;
–
 Variable Declarations
var Look: char;  Lookahead Character
–
 Read New Character From Input Stream
procedure GetChar;
begin
Read(Look);
end;
–
 Report an Error
procedure Error(s: string);
begin
WriteLn;
WriteLn(^G, 'Error: ', s, '.');
end;
–
 Report Error and Halt
procedure Abort(s: string);
begin
Error(s);
Halt;
end;
–
 Report What Was Expected
procedure Expected(s: string);
begin
Abort(s + ' Expected');
end;
–
 Match a Specific Input Character
procedure Match(x: char);
begin
if Look = x then GetChar
else Expected("" + x + "");
end;
–
 Recognize an Alpha Character
function IsAlpha(c: char): boolean;
begin
IsAlpha := upcase(c) in ['A'..'Z'];
end;
–
 Recognize a Decimal Digit
function IsDigit(c: char): boolean;
begin
IsDigit := c in ['0'..'9'];
end;
–
 Get an Identifier
function GetName: char;
begin
if not IsAlpha(Look) then Expected('Name');
GetName := UpCase(Look);
GetChar;
```

```
end;
–
 Get a Number
function GetNum: char;
begin
if not IsDigit(Look) then Expected('Integer');
GetNum := Look;
GetChar;
end;
–
 Output a String with Tab
procedure Emit(s: string);
begin
Write(TAB, s);
end;
–
 Output a String with Tab and CRLF
procedure EmitLn(s: string);
begin
Emit(s);
WriteLn;
end;
–
 Initialize
procedure Init;
begin
GetChar;
end;
–
 Main Program
begin
Init;
end.
–
```

Скопируйте код, представленный выше, в TP и откомпилируйте. Удостоверьтесь, что программа откомпилировалась и запустилась корректно. Затем переходим к первому уроку, синтаксическому анализу выражений.

Синтаксический анализ выражений

НАЧАЛО

Если вы прочитали введение, то вы уже в курсе дела. Вы также скопировали программу Cradle в Turbo Pascal и откомпилировали ее. Итак, вы готовы.

Целью этой главы является обучение синтаксическому анализу и трансляции математических выражений. В результате мы хотели бы видеть серию команд на ассемблере, выполняющую необходимые действия. Выражение – правая сторона уравнения, например:

x = 2*y + 3/(4*z)

В самом начале я буду двигаться очень маленькими шагами для того, чтобы начинающие из вас совсем не заблудились. Вы также получите несколько хороших уроков, которые хорошо послужат нам позднее. Для более опытных читателей: потерпите. Скоро мы двинемся вперед.

ОДИНОЧНЫЕ ЦИФРЫ

В соответствии с общей темой этой серии (KISS-принцип, помнишь?), начнем с самого простого случая, который можно себе представить. Это выражение, состоящее из одной цифры.

Перед тем как начать, удостоверьтесь, что у вас есть базовая копия Cradle. Мы будем использовать ее для других экспериментов. Затем добавьте следующие строки:

```
{--------------------------------------------------------------}
{ Parse and Translate a Math Expression }
procedure Expression;
begin
EmitLn('MOVE #' + GetNum + ',D0')
end;
{--------------------------------------------------------------}
```

И добавьте строку “Expression;” в основную программу, которая должна выглядеть так:

```
{--------------------------------------------------------------}
begin
Init;
Expression;
end.
{--------------------------------------------------------------}
```

Теперь запустите программу. Попробуйте ввести любую одиночную цифру. Вы получите результат в виде одной строчки на ассемблере. Затем попробуйте ввести любой другой символ и вы увидите, что синтаксический анализатор правильно сообщает об ошибке.

Поздравляю! Вы только что написали работающий транслятор!

Конечно, я понимаю, что он очень ограничен. Но не отмахивайтесь от него. Этот маленький «компилятор» в ограниченных масштабах делает точно то же, что делает любой большой компилятор: он корректно распознает допустимые утверждения на входном «языке», который мы для него определили, и производит корректный, выполнимый ассемблерный код, пригодный для перевода в объектный формат. И, что важно, корректно распознает недопустимые утверждения, выдавая сообщение об ошибке. Кому требовалось больше?

Имеются некоторые другие особенности этой маленькой программы, заслуживающие внимания. Во первых, вы видите, что мы не отделяем генерацию кода от синтаксического анализа… как только анализатор узнает что нам нужно, он непосредственно генерирует объектный код. В настоящих компиляторах, конечно, чтение в GetChar должно происходить из файла и затем выполняться запись в другой файл, но этот способ намного проще пока мы экспериментируем.

Также обратите внимание, что выражение должно где-то сохранить результат. Я выбрал регистр D0 процессора 68000. Я мог бы выбрать другой регистр, но в данном случае это имеет смысл.

ВЫРАЖЕНИЯ С ДВУМЯ ЦИФРАМИ

Теперь, давайте немного улучшим то, что у нас есть. По общему признанию, выражение, состоящее только из одного символа, не удовлетворит наших потребностей надолго, так что давайте посмотрим, как мы можем расширить возможности компилятора. Предположим, что мы хотим обрабатывать выражения вида:

1+2

или 4-3

или в общем <term> +/– <term> (это часть формы Бэкуса-Наура или БНФ.)

Для того, чтобы сделать это, нам нужна процедура, распознающая термы и сохраняющая результат, и другая процедура, которая распознает и различает «+» и «-» и генерирует соответствующий код. Но если процедура Expression сохраняет свои результаты в регистре D0, то где процедура Term сохранит свои результаты? Ответ: на том же месте. Мы окажемся перед необходимостью сохранять первый результат процедуры Term где-нибудь, прежде чем мы получим следующий.

В основном, что нам необходимо сделать – создать процедуру Term, выполняющую то что раннее выполняла процедура Expression. Поэтому просто переименуйте процедуру Expression в Term и наберите новую версию Expression:

```
{--------------------------------------------------------------}
{ Parse and Translate an Expression }
procedure Expression;
begin
Term;
EmitLn('MOVE D0,D1');
case Look of
'+': Add;
'-': Subtract;
else Expected('Addop');
end;
end;
{--------------------------------------------------------------}
```

Затем выше Expression наберите следующие две процедуры:

```
{--------------------------------------------------------------}
{ Recognize and Translate an Add }
procedure Add;
begin
Match('+');
Term;
EmitLn('ADD D1,D0');
end;
{--------------------------------------------------------------}
{ Recognize and Translate a Subtract }
procedure Subtract;
begin
Match('-');
Term;
EmitLn('SUB D1,D0');
end;
{--------------------------------------------------------------}
```

Когда вы закончите, порядок подпрограмм должен быть следующий:

Term (старая версия Expression)

Add

Subtract

Expression

Теперь запустите программу. Испробуйте любую комбинацию, которую вы только можете придумать, из двух одиночных цифр, разделенных «+» или «-». Вы должны получить ряд из четырех инструкций на ассемблере. Затем испытайте выражения с заведомыми ошибками в них. Перехватывает анализатор ошибки?

Посмотрите на полученный объектный код. Можно сделать два замечания. Во первых, сгенерированный код не такой, какой бы написали мы. Последовательность

MOVE #n,D0
MOVE D0,D1

неэффективна. Если бы мы писали этот код вручную, то, возможно, просто загрузили бы данные напрямую в D1.

Вывод: код, генерируемый нашим синтаксическим анализатором, менее эффективный, чем код, написанный вручную. Привыкните к этому. Это в известной мере относится ко всем компиляторам. Ученые посвятили целые жизни вопросу оптимизации кода и существуют методы, призванные улучшить качество генерируемого кода. Некоторые компиляторы выполняют оптимизацию достаточно хорошо, но за это приходится платить сложностью и в любом случае это проигранная битва… возможно никогда не придет время, когда хороший программист на ассемблере не смог бы превзойти компилятор. Прежде чем закончится этот урок, я кратко упомяну некоторые способы, которые мы можем применить для небольшой оптимизации, просто, чтобы показать вам, что мы на самом деле сможем сделать некоторые улучшения без излишних проблем. Но запомните, мы здесь для того, чтобы учиться, а не для того, чтобы узнать насколько компактным мы можем сделать код. А сейчас и на протяжении всей этой серии мы старательно будем игнорировать оптимизацию и сконцентрируемся на получении работающего кода.

Но наш код не работает! В коде есть ошибка! Команда вычитания вычитает D1 (первый аргумент) из D0 (второй аргумент). Но это неправильный способ, так как мы получаем неправильный знак результата. Поэтому исправим процедуру Subtract с помощью замены знака следующим образом:

```
_
 Recognize and Translate a Subtract
procedure Subtract;
begin
Match('-');
Term;
EmitLn('SUB D1,D0');
EmitLn('NEG D0');
end;
_
```

Теперь наш код даже еще менее эффективен, но по крайней мере выдает правильный ответ! К сожалению, правила, которые определяют значение математических выражений, требуют, чтобы условия в выражении следовали в неудобном для нас порядке. Опять, это только один из фактов жизни, с которыми вы учитесь жить. Все это возвратится снова, чтобы преследовать нас, когда мы примемся за деление.

Итак, на данном этапе мы имеем синтаксический анализатор, который может распознавать сумму или разность двух цифр. Ранее мы могли распознавать только одиночные цифры. Но настоящие выражения могут иметь любую форму (или бесконечность других). Вернитесь и запустите программу с единственным входным символом “1”.

Не работает? А почему должен работать? Мы только указали анализатору, что единственным правильными видами выражений являются выражения с двумя термами. Мы должны переписать процедуру Expression так, чтобы она была намного более универсальной и с этого начать создание настоящего синтаксического анализатора.

ОБЩАЯ ФОРМА ВЫРАЖЕНИЯ

В реальном мире выражение может состоять из одного или более термов, разделенных «addops» ('+' или '-'). В БНФ это может быть записано как:

<expression> ::= <term> [<addop> <term>]*

Мы можем применить это определение выражения, добавив простой цикл к процедуре

Expression:

```
–
 Parse and Translate an Expression
procedure Expression;
begin
Term;
while Look in ['+', '-'] do begin
EmitLn('MOVE D0,D1');
case Look of
'+': Add;
'-': Subtract;
else Expected('Addop');
end;
end;
end;
–
```

Эта версия поддерживает любое число термов, и это стоило нам только двух дополнительных строк кода. По мере изучения, вы обнаружите, что это характерно для нисходящих синтаксических анализаторов… необходимо только несколько дополнительных строк кода чтобы добавить расширения языка. Это как раз то, что делает наш пошаговый метод возможным. Заметьте также, как хорошо код процедуры Expression соответствует определению БНФ. Это также одна из характеристик метода. Когда вы станете специалистом этого метода, вы сможете превращать БНФ в код синтаксического анализатора примерно с такой же скоростью, с какой вы можете набирать текст на клавиатуре!

ОК, откомпилируйте новую версию анализатора и испытайте его. Как обычно, проверьте что «компилятор» обрабатывает любое допустимое выражение и выдает осмысленное сообщение об ошибке для запрещенных. Четко, да? Вы можете заметить, что в нашей тестовой версии любое сообщение об ошибке выводится вместе с генерируемым кодом. Но запомните, это только потому, что мы используем экран как «выходной файл» в этих экспериментах. В рабочей версии вывод будет разделен… один в выходной файл, другой на экран.

ИСПОЛЬЗОВАНИЕ СТЕКА

В этом месте я собираюсь нарушить свое правило, что я не представлю что-либо сложное, пока это не будет абсолютно необходимо. Прошло достаточно много времени, чтобы не отметить проблему с генерируемым кодом. В настоящее время синтаксический анализатор использует D0 как «основной» регистр, и D1 для хранения частичной суммы. Эта схема работает отлично потому что мы имеем дело только с «addops» (“+” и “-”) и новое число прибавляется по мере появления. Но в общем форме это не так. Рассмотрим, например выражение

1+(2-(3+(4-5)))

Если мы поместим «1» в D1, то где мы разместим «2»? Так как выражение в общей форме может иметь любую степень сложности, то мы очень быстро используем все регистры!

К счастью есть простое решение. Как и все современные микропроцессоры, 68000 имеет стек, который является отличным местом для хранения переменного числа элементов. Поэтому вместо того, чтобы помещать термы в D0 и D1 давайте затолкнем их в стек. Для тех кто незнаком с ассемблером 68000 – помещение в стек пишется как

–(SP)

и извлечение (SP)+.

Итак, изменим EmitLn в процедуре Expression на

EmitLn('MOVE D0,-(SP)');
и две строки в Add и Subtract:
EmitLn('ADD (SP)+,D0') и EmitLn('SUB (SP)+,D0')
соответственно. Теперь испытаем компилятор снова и удостоверимся что он работает.
И снова, полученный код менее эффективен, чем был до этого, но это необходимый шаг, как вы увидите.

УМНОЖЕНИЕ И ДЕЛЕНИЕ

Теперь давайте возьмемся за действительно серьезные дела. Как вы знаете, кроме операторов «addops» существуют и другие… выражения могут также иметь операторы умножения и деления. Вы также знаете, что существует неявный приоритет операторов или иерархия, связанная с выражениями, чтобы в выражениях типа
2 + 3 * 4,
мы знали, что нужно сначала умножить, а затем сложить. (Видите, зачем нам нужен стек?)
В ранние дни технологии компиляторов, люди использовали различные довольно сложные методы для того чтобы правила приоритета операторов соблюдались. Но, оказывается, все же, что ни один из них нам не нужен… эти правила могут быть очень хорошо применены в нашей технике нисходящего синтаксического анализа. До сих пор единственной формой, которую мы применяли для терма была форма одиночной десятичной цифры. В более общей форме мы можем определить терм как произведение показателей (product of factors), то есть
<term> ::= <factor> [<mulop> <factor]*
Что такое показатель? На данный момент это тоже, чем был раннее терм – одиночной цифрой.
Обратите внимание: терм имеет ту же форму, что и выражение. Фактически, мы можем добавить это в наш компилятор осторожно скопировав и переименовав. Но во избежание неразберихи ниже приведен полный листинг всех подпрограмм анализатора. (Заметьте способ, которым мы изменяем порядок операндов в Divide.)

```
{--------------------------------------------------------------}
{ Parse and Translate a Math Factor }
procedure Factor;
begin
EmitLn('MOVE #' + GetNum + ',D0')
end;
{--------------------------------------------------------------}
{ Recognize and Translate a Multiply }
procedure Multiply;
begin
Match('*');
Factor;
EmitLn('MULS (SP)+,D0');
end;
{--------------------------------------------------------------}
{ Recognize and Translate a Divide }
procedure Divide;
begin
Match('/');
Factor;
EmitLn('MOVE (SP)+,D1');
EmitLn('DIVS D1,D0');
end;
```

```
{
 Parse and Translate a Math Term
procedure Term;
begin
Factor;
while Look in ['*', '/'] do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'*': Multiply;
'/': Divide;
else Expected('Mulop');
end;
end;
end;
{
 Recognize and Translate an Add
procedure Add;
begin
Match('+');
Term;
EmitLn('ADD (SP)+,D0');
end;
{
 Recognize and Translate a Subtract
procedure Subtract;
begin
Match('-');
Term;
EmitLn('SUB (SP)+,D0');
EmitLn('NEG D0');
end;
{
 Parse and Translate an Expression
procedure Expression;
begin
Term;
while Look in ['+', '-'] do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'+': Add;
'-': Subtract;
else Expected('Addop');
end;
end;
end;
{
```

Конфетка! Почти работающий транслятор в 55 строк Паскаля! Получаемый код начинает выглядеть действительно полезным, если не обращать внимание на неэффективность. Запомните, мы не пытаемся создавать сейчас самый компактный код.

КРУГЛЫЕ СКОБКИ

Мы можем закончить эту часть синтаксического анализатора добавив поддержку круглых скобок. Как вы знаете, скобки являются механизмом принудительного изменения приоритета операторов. Так, например, в выражении

2*(3+4) ,

скобки заставляют выполнять сложение перед умножением. Но, что гораздо более важно, скобки дают нам механизм для определения выражений любой степени сложности, как, например

(1+2)/((3+4)+(5-6))

Ключом к встраиванию скобок в наш синтаксический анализатор является понимание того, что не зависимо от того, как сложно выражение, заключенное в скобки, для остальной части мира оно выглядит как простой показатель. Это одна из форм для показателя:

<factor> ::= (<expression>)

Здесь появляется рекурсия. Выражение может содержать показатель, который содержит другое выражение, которое содержит показатель и т.д. до бесконечности.

Сложно это или нет, мы должны позаботиться об этом, добавив несколько строчек в процедуру Factor:

```
–
 Parse and Translate a Math Factor
procedure Expression; Forward;
procedure Factor;
begin
if Look = '(' then begin
Match('(');
Expression;
Match(')');
end
else
EmitLn('MOVE #' + GetNum + ',D0');
end;
–
```

Заметьте снова, как легко мы можем дополнять синтаксический анализатор, и как хорошо код Паскаля соответствует синтаксису БНФ.

Как обычно, откомпилируйте новую версию и убедитесь, что анализатор корректно распознает допустимые предложения и отмечает недопустимые сообщениями об ошибках.

УНАРНЫЙ МИНУС

На данном этапе мы имеем синтаксический анализатор, который поддерживает почти любые выражения, правильно? ОК, тогда испробуйте следующее предложение:

–1

Опс! Он не работает, не правда ли? Процедура Expression ожидает, что все числа будут целыми и спотыкается на знаке минус. Вы найдете, что +3 также не будет работать, так же как и что-нибудь типа:

–(3-2).

Существует пара способов для исправления этой проблемы. Самый легкий (хотя и не обязательно самый лучший) способ – вставить ноль в начало выражения, так чтобы -3 стал 0-3. Мы можем легко исправить это в существующей версии Expression:

```
–
 Parse and Translate an Expression
procedure Expression;
begin
if IsAddop(Look) then
EmitLn('CLR D0')
else
```

```
Term;
while IsAddop(Look) do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'+': Add;
'-': Subtract;
else Expected('Addop');
end;
end;
end;
_
```

Я говорил вам, насколько легко мы сможем вносить изменения! На этот раз они стоили нам всего трех новых строчек Паскаля. Обратите внимание на появление ссылки на новую функцию IsAddop. Как только проверка на addop появилась дважды, я решил выделить ее в отдельную функцию. Форма функции IsAddop должна быть аналогична форме функции IsAlpha. Вот она:

```
_
 Recognize an Addop
function IsAddop(c: char): boolean;
begin
IsAddop := c in ['+', '-'];
end;
_
```

ОК, внесите эти изменения в программу и повторно откомпилируйте. Вы должны также включить IsAddop в базовую копию программы Cradle. Она потребуется нам позже. Сейчас попробуйте снова ввести -1. Вау! Эффективность полученного кода довольно плохая… шесть строк кода только для того, чтобы загрузить простую константу… но, по крайней мере, правильно работает. Запомните, мы не пытаемся сделать замену Turbo Pascal.

На данном этапе мы почти завершили создание структуры нашего синтаксического анализатора выражений. Эта версия программы должна правильно распознавать и компилировать почти любое выражение, которое вы ей подсунете. Она все еще ограничена тем, что поддерживает показатели состоящие только из одной цифры. Но я надеюсь что теперь вы начинаете понимать, что мы можем расширять возможности синтаксического анализатора делая незначительные изменения. Вы возможно даже не будете удивлены, когда услышите, что переменная или даже вызов функции это просто один из видов показателя.

В следующей главе я покажу, как можно легко расширить наш синтаксический анализатор для поддержки всех этих возможностей, и я также покажу как легко мы можем добавить многосимвольные числа и имена переменных. Итак, вы видите, что мы совсем недалеко от действительно полезного синтаксического анализатора.

СЛОВО ОБ ОПТИМИЗАЦИИ

Раннее в этой главе я обещал дать несколько подсказок как мы можем повысить качество генерируемого кода. Как я сказал, получение компактного кода не является главной целью этой книги. Но вам нужно по крайней мере знать, что мы не зря проводим свое время… что мы действительно можем модифицировать анализатор для получения лучшего кода не выбрасывая то, что мы уже сделали к настоящему времени. Обычно небольшая оптимизация не слишком трудна… просто в синтаксический анализатор вставляется дополнительный код.

Существуют два основных метода, которые мы можем использовать:

Попытаться исправить код после того, как он сгенерирован.

Это понятие «щелевой» оптимизации. Основная идея в том, что известно какие комбинации инструкций компилятор собирается произвести и также известно которые из них «плохие» (такие как код для числа -1). Итак, все что нужно сделать – просканировать полученный код, найти такие комбинации инструкций и заменить их на более «хорошие». Это вид макрорасширений наоборот и прямой пример метода сопоставления с образцом. Единственная сложность в том, что может существовать множество таких комбинаций. Этот метод называется «щелевой» оптимизацией просто потому, что оптимизатор работает с маленькой группой инструкций. «Щелевая» оптимизация может драматически влиять на качество кода и не требует при этом больших изменений в структуре компилятора. Но все же за это приходится платить скоростью, размером и сложностью компилятора. Поиск всех комбинаций требует проверки множества условий, каждая из которых является источником ошибки. И, естественно, это требует много времени.

В классической реализации «щелевого» оптимизатора, оптимизация выполняется как второй проход компилятора. Выходной код записывается на диск и затем оптимизатор считывает и обрабатывает этот файл снова. Фактически, оптимизатор может быть даже отдельной от компилятора программой. Так как оптимизатор только обрабатывает код в маленьком «окне» инструкций (отсюда и название), лучшей реализацией было бы буферизировать несколько срок выходного кода и сканировать буфер каждый раз после EmitLn.

Попытаться сразу генерировать лучший код.

В этом методе выполняется проверка дополнительных условий перед выводом кода. Как тривиальный пример, мы должны были бы идентифицировать нуль и выдать CLR вместо загрузки, или даже совсем ничего не делать, как в случае с прибавлением нуля, например. Конкретней, если мы решили распознавать унарный минус в процедуре Factor вместо Expression, то мы должны обрабатывать –1 как обычную константу, а не генерировать ее из положительных. Ни одна из этих вещей не является слишком сложной для реализации… просто они требуют включения дополнительных проверок в код, поэтому я не включил их в программу. Как только мы дойдем до получения работающего компилятора, генерирующего полезный выполнимый код, мы всегда сможем вернуться и доработать программу для получения более компактного кода. Именно поэтому в мире существует «Версия 2.0».

Существует еще один, достойный упоминания, способ оптимизации, обещающий достаточно компактный код без излишних хлопот. Это мое «изобретение», в том смысле, что я нигде не видел публикаций по этому методу, хотя я и не питаю иллюзий что это придумано мной.

Способ заключается в том, чтобы избежать частого использования стека, лучше используя регистры центрального процессора. Вспомните, когда мы выполняли только сложение и вычитание, то мы использовали регистры D0 и D1 а не стек? Это работало, потому для этих двух операций стек никогда не использовал более чем две ячейки.

Хорошо, процессор 68000 имеет восемь регистров данных. Почему бы не использовать их как стек? В любой момент своей работы синтаксический анализатор «знает» как много элементов в стеке, поэтому он может правильно ими манипулировать. Мы можем определить частный указатель стека, который следит, на каком уровне мы находимся и адресует соответствующий регистр. Процедура Factor, например, должна загружать данные не в регистр D0, а в тот, который является текущей вершиной стека.

Что мы получаем заменяя стек в RAM на локальный стек созданный из регистров. Для большинства выражений уровень стека никогда не превысит восьми, поэтому мы получаем достаточно хороший код. Конечно, мы должны предусмотреть те случаи, когда уровень стека превысит восемь, но это также не проблема. Мы просто позволим стеку перетекать в стек ЦПУ. Для уровней выше восьми код не хуже, чем тот, который мы генерируем сейчас, а для уровней ниже восьми он значительно лучше.

Я реализовал этот метод, просто для того, чтобы удостовериться в том, что он работает

перед тем, как представить его вам. Он работает. На практике вы не можете в действительности использовать все восемь уровней... вам, как минимум, нужен один свободный регистр для изменения порядка операндов при делении. Для выражений, включающих вызовы функций, также необходимо зарезервировать регистр. Но все равно, существует возможность улучшения размера кода для большинства выражений.

Итак, вы видите, что получение лучшего кода не настолько трудно, но это усложняет наш транслятор... это сложность, без которой мы можем сейчас обойтись. По этой причине, я очень советую продолжать игнорировать вопросы эффективности в этой книге, усвоив, что мы действительно можем повысить качество кода не выбрасывая того, что уже сделано.

В следующей главе я покажу вам как работать с переменными и вызовами функций. Я также покажу вам как легко добавить поддержку многосимвольных токенов и пробелов.

Снова выражения

ВВЕДЕНИЕ

В последней главе мы изучили методы, используемые для синтаксического анализа и трансляции математических выражений в общей форме. Мы закончили созданием простого синтаксического анализатора, поддерживающего выражения произвольной сложности с двумя ограничениями:

Разрешены только числовые показатели

Числовые показатели ограничены одиночной цифрой.

В этой главе мы избавимся от этих ограничений. Мы также расширим то что сделали, добавив операции присваивания и вызовы функций. Запомните, однако, что второе ограничение было главным образом наложено нами самими... выбрано для удобства, чтобы облегчить себе жизнь и сконцентрироваться на фундаментальных принципах. Как вы увидите, от этого ограничения легко освободиться, так что не слишком задерживайтесь на этом. Мы будем использовать это прием пока он служит нам, уверенные в том, что сможем избавиться от него, когда будем готовы.

ПЕРЕМЕННЫЕ

Большинство выражений, который мы встречаем на практике, включают переменные, например:

b * b + 4 * a * c

Ни один компилятор нельзя считать достаточно хорошим, если он не работает с ними. К счастью, это тоже очень просто сделать.

Не забудьте, что в нашем синтаксическом анализаторе в настоящее время существуют два вида показателей: целочисленные константы и выражения в скобках. В нотации БНФ:

<factor> ::= <number> | (<expression>)

"|" заменяет «или», означая, что любая из этих форм является допустимой. Запомните, также, что у нас нет проблемы в определении каждой их них… предсказывающим символом в одном случае является левая скобка "(" и цифра – в другом.

Возможно, не вызовет слишком большого удивления то, что переменная – это просто еще один вид показателя. Так что расширим БНФ следующим образом:

<factor> ::= <number> | (<expression>) | <variable>

И снова, здесь нет неоднозначности: если предсказывающий символ – буква, то это переменная, если цифра то число. Когда мы транслируем число, мы просто генерируем код для загрузки числа, как промежуточных данных, в D0. Сейчас мы делаем то же самое, только для переменной.

Небольшое осложнение при генерации кода возникает из того факта, что большинство

операционных систем для 68000, включая SK*DOS которую я использую, требуют чтобы код был написан в «переместимой» форме, что в основном означает что все должно быть PC-относительно. Формат для загрузки на этом языке будет следующим:

MOVE X(PC),D0

где X, конечно, имя переменной. Вооружившись этим, изменим текущую версию процедуры Factor следующим образом:

```
{
 Parse and Translate a Math Factor
procedure Expression; Forward;
procedure Factor;
begin
if Look = '(' then begin
Match('(');
Expression;
Match(')');
end
else if IsAlpha(Look) then
EmitLn('MOVE ' + GetName + '(PC),D0')
else
EmitLn('MOVE #' + GetNum + ',D0');
end;
}
```

Я уже отмечал, как легко добавлять расширения в синтаксический анализатор благодаря способу его структурирования. Вы можете видеть, что это все еще остается действительным и сейчас. На этот раз это стоило нам всего двух дополнительных строк кода. Заметьте так же, как структура if-then-else точно соответствует синтаксическому уравнению БНФ.

ОК, откомпилируйте и протестируйте эту новую версию синтаксического анализатора. Это не слишком сильно повредило, не так ли?

ФУНКЦИИ

Есть еще только один распространенный вид показателей, поддерживаемый большинством языков: вызов функции. В действительности, нам пока слишком рано иметь дела с функциями, потому что мы еще не обращались к вопросу передачи параметров. Более того, «настоящий» язык должен включать механизм поддержки более чем одного типа, одним из которых должен быть тип функции. Мы не имеем также и этого. Но все же я хотел бы работать с функциями сейчас по двум причинам. Во-первых, это позволит нам превратить компилятор во что-то очень близкое к конечному виду и, во вторых, это раскроет новую проблему, о которой очень стоит поговорить.

До этого момента мы создавали то, что называется «предсказывающим синтаксическим анализатором». Это означает, что в любой точке мы можем, смотря на текущий предсказывающий символ, точно знать, что будет дальше. Но не в том случае когда мы добавляем функции. В каждом языке имеются некоторые правила присваивания имен, по которым составляется допустимый идентификатор. Наши правила пока просты, так как идентификатором является одна из букв "a"…"z". Проблема состоит в том, что имена переменных и имена функций подчиняются одним и тем же правилам. Поэтому как мы можем сказать кто из них кто? Один из способов требует, чтобы каждое из них было объявлено перед тем, как оно используется. Этот метод использует Pascal. Другой способ состоит в том, чтобы функция сопровождалась списком параметров (возможно пустым). Это правило, используемое в C.

Пока у нас нет механизма описания типов, давайте использовать правила C. Так как у

нас также нет и механизма для работы с параметрами, мы можем поддерживать только пустые списки параметров, так что вызовы функций будут иметь следующую форму:

X().

Так как мы пока не работаем со списками параметров, для вызова функций не нужно ничего дополнительно, и необходимо только выдавать BSR (вызов) вместо MOVE.

Сейчас существуют две варианта для ветки «If IsAlpha» при проверке в процедуре Factor. Давайте обработаем их в отдельной процедуре. Изменим процедуру Factor следующим образом:

```
{--------------------------------------------------------------}
{ Parse and Translate a Math Factor }
procedure Expression; Forward;
procedure Factor;
begin
if Look = '(' then begin
Match('(');
Expression;
Match(')');
end
else if IsAlpha(Look) then
Ident
else
EmitLn('MOVE #' + GetNum + ',D0');
end;
{--------------------------------------------------------------}
```

и вставим перед ней новую процедуру

```
{--------------------------------------------------------------}
{ Parse and Translate an Identifier }
procedure Ident;
var Name: char;
begin
Name := GetName;
if Look = '(' then begin
Match('(');
Match(')');
EmitLn('BSR ' + Name);
end
else
EmitLn('MOVE ' + Name + '(PC),D0')
end;
{--------------------------------------------------------------}
```

Откомпилируйте и протестируйте эту версию. Обрабатывает ли она все правильные выражения и корректно отмечает неправильные?

Важно отметить, что хотя наш анализатор больше не является предсказывающим анализаторов, это немного или совсем не добавляет сложностей при использовании нами метода рекурсивного спуска. В том месте, где процедура Factor находит идентификатор (букву), она не знает, является ли он именем переменной или именем функции, ни выполняет ее обработку. Она просто передает его в Ident и оставляет этой процедуре на рассмотрение. Ident, в свою очередь, просто прячет идентификатор и затем считывает еще один символ для того, чтобы решить с каким типом идентификатора он имеет дело.

Запомните этот способ. Это очень мощное понятие и оно должно быть использовано всегда, когда вы встречаетесь с неоднозначной ситуацией, требующей заглядывания вперед.

Даже если вам нужно рассмотреть несколько символов вперед, принцип все еще будет работать.

ПОДРОБНЕЕ ОБ ОБРАБОТКЕ ОШИБОК

Имеется еще одна важная проблема, которую стоит отметить: обработка ошибок. Обратите внимание, что хотя синтаксический анализатор правильно отбрасывает (почти) каждое некорректное выражение, которое мы ему подбросим, со значимым сообщением об ошибке, в действительности мы не слишком много поработали для того, чтобы это происходило. Фактически во всей программе (от Ident до Expression) есть только два вызова подпрограммы обработки ошибок Expected. Но даже они не являются необходимыми… если вы посмотрите снова на процедуры Term и Expression, то увидите, что эти утверждения не выполнятся никогда. Я поместил их сюда ранее для небольшой подстраховки, но сейчас они более не нужны. Почему бы не удалить их сейчас?

Но как мы получали такую хорошую обработку ошибок фактически бесплатно? Просто я тщательно старался избежать чтения символа непосредственно используя GetChar. Взамен я возложил на GetName, GetNum и Match выполнение всей обработки ошибок для меня. Проницательные читатели заметят, что некоторые вызовы Match (к примеру в Add и Subtract) также не нужны… мы уже знаем чем является символ к этому времени… но их присутствие сохраняет некоторую симметрию, и было бы хорошим правилом всегда использовать Match вместо GetChar.

Выше я упомянул «почти». Есть случай, когда наша обработка ошибок оставляет желать лучшего. Пока что мы не сказали нашему синтаксическому анализатору как выглядит конец строки или что делать с вложенными пробелами. Поэтому пробел (или любой другой символ, не являющийся частью признаваемого набора символов) просто вызывает завершение работы анализатора, игнорируя нераспознанные символы.

Можно рассудить, что в данном случае это приемлемое поведение. В «настоящем» компиляторе обычно присутствует еще одно утверждение, следующее после того, с которым мы работаем, так что любой символ, не обработанный как часть нашего выражения, будет или использоваться или отвергаться как часть следующего.

Но это также очень легко исправить, даже если это только временно. Все, что мы должны сделать – постановить, что выражение должно заканчиваться концом строки, то есть, возвратом каретки.

Чтобы понять о чем я говорю, испробуйте входную строку:

1+2 <space> 3+4

Видите, как пробел был обработан как признак завершения? Чтобы заставить компилятор правильно отмечать это, добавьте строку

if Look <> CR then Expected('Newline');

в основную программу, сразу после вызова Expression. Это отлавливает все левое во входном потоке. Не забудьте определить CR в разделе const:

CR = ^M;

Как обычно откомпилируйте программу и проверьте, что она делает то, что нужно.

ПРИСВАИВАНИЕ

Итак, к этому моменту мы имеем синтаксический анализатор, работающий очень хорошо. Я хотел бы подчеркнуть, что мы получили это, используя всего 88 строк выполнимого кода, не считая того, что было в Cradle. Откомпилированный объектный файл занял 4752 байта. Неплохо, учитывая то, что мы не слишком старались сохранять размеры как исходного так и объектного кода. Мы просто придерживались принципа KISS.

Конечно, анализ выражений не настолько хорош без возможности что-либо делать с его результатами. Выражения обычно (но не всегда) используются в операциях

присваивания в форме:

<Ident> = <Expression>

Мы находимся на расстоянии вздоха от возможности анализировать операции присваивания, так что давайте сделаем этот последний шаг. Сразу после процедуры Expression добавьте следующую новую процедуру:

```
_
 Parse and Translate an Assignment Statement
procedure Assignment;
var Name: char;
begin
Name := GetName;
Match('=');
Expression;
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE D0,(A0)')
end;
_
```

Обратите внимание снова, что код полностью соответствует БНФ. И заметьте затем, что проверка ошибок была безболезненна и обработана GetName и Match.

Необходимость двух строк ассемблера возникает из-за особенности 68000, который требует такого вида конструкции для PC-относительного кода.

Теперь измените вызов Expression в основной программе на Assigment. Это все, что нужно.

Фактически мы компилируем операторы присваивания! Если бы это был единственный вид операторов в языке, все, что нам нужно было бы сделать – поместить его в цикл и мы имели бы полноценный компилятор!

Конечно, это не единственный вид. Есть также немного таких элементов, как управляющие структуры (ветвления и циклы), процедуры, объявления и т.д. Но не унывайте. Арифметические выражения, с которыми мы имели дело, относятся к самым вызывающим элементам языка. По сравнению с тем, что мы уже сделали, управляющие структуры будут выглядеть простыми. Я расскажу о них в пятой главе. И все другие операторы поместятся в одной строчке, пока мы не забываем принцип KISS.

МНОГОСИМВОЛЬНЫЕ ТОКЕНЫ.

В этой серии я тщательно ограничивал все, что мы делаем, односимвольными токенами, все время уверяя вас, что не составит проблемы расширить их до многосимвольных. Я не знаю, верили вы мне или нет… я действительно не обвинил бы вас, если бы вы были немного скептичны. Я буду продолжать использовать этот подход и в следующих главах, потому что это позволит избежать сложности. Но я хотел бы поддержать эту уверенность и показать вам, что это действительно легко сделать. В процессе этого мы также предусмотрим обработку вложенных пробелов. Прежде чем вы сделаете следующие несколько изменений, сохраните текущую версию синтаксического анализатора под другим именем. Я буду использовать ее в следующей главе и мы будем работать с односимвольной версией.

Большинство компиляторов выделяют обработку входного потока в отдельный модуль, называемый лексическим анализатором (сканером). Идея состоит в том, что сканер работает со всей последовательностью символов во входном потоке и возвращает отдельные единицы (лексемы) потока. Возможно придет время, когда мы также захотим сделать что-то вроде этого, но сейчас в этом нет необходимости. Мы можем обрабатывать многосимвольные токены, которые нам нужны, с помощью небольших локальных изменений в GetName и

GetNum.

Обычно признаком идентификатора является то, что первый символ должен быть буквой, но остальная часть может быть алфавитно-цифровой (буквы и цифры). Для работы с ними нам нужна другая функция:

```
{--------------------------------------------------------------}
{ Recognize an Alphanumeric }
function IsAlNum(c: char): boolean;
begin
IsAlNum := IsAlpha(c) or IsDigit(c);
end;
{--------------------------------------------------------------}
```

Добавьте эту функцию в анализатор. Я поместил ее сразу после IsDigit. Вы можете также включить ее как постоянного члена в Cradle.

Теперь нам необходимо изменить функцию GetName так, чтобы она возвращала строку вместо символа:

```
{--------------------------------------------------------------}
{ Get an Identifier }
function GetName: string;
var Token: string;
begin
Token := '';
if not IsAlpha(Look) then Expected('Name');
while IsAlNum(Look) do begin
Token := Token + UpCase(Look);
GetChar;
end;
GetName := Token;
end;
{--------------------------------------------------------------}
```

Аналогично измените GetNum следующим образом:

```
{--------------------------------------------------------------}
{ Get a Number }
function GetNum: string;
var Value: string;
begin
Value := '';
if not IsDigit(Look) then Expected('Integer');
while IsDigit(Look) do begin
Value := Value + Look;
GetChar;
end;
GetNum := Value;
end;
{--------------------------------------------------------------}
```

Достаточно удивительно, что это фактически все необходимые изменения! Локальная переменная Name в процедурах Ident и Assignment были первоначально объявлены как «char» и теперь должны быть объявлены как string[8]. (Ясно, что мы могли бы сделать длину строки больше, если бы захотели, но большинство ассемблеров в любом случае ограничивают длину.) Внесите эти изменения и затем откомпилируйте и протестируйте.

Сейчас вы верите, что это просто?

ПРОБЕЛЫ

Прежде, чем мы оставим этот синтаксический анализатор на некоторое время, давайте обратимся к проблеме пробелов. На данный момент, синтаксический анализатор выразит недовольство (или просто завершит работу) на одиночном символе пробела, вставленном где-нибудь во входном потоке. Это довольно недружелюбное поведение. Так что давайте немного усовершенствуем анализатор, избавившись от этого последнего ограничения.

Ключом к облегчению обработки пробелов является введение простого правила для того, как синтаксический анализатор должен обрабатывать входной поток и использование этого правила везде. До настоящего времени, поскольку пробелы не были разрешены, у нас была возможность знать, что после каждого действия синтаксического анализатора предсказывающий символ Look содержит следующий значимый символ, поэтому мы могли немедленно выполнять его проверку. Наш проект был основан на этом принципе.

Это все еще звучит для меня как хорошее правило, поэтому мы будем его использовать. Это означает, что каждая подпрограмма, которая продвигает входной поток, должна пропустить пробелы и оставить следующий символ (не являющийся пробелом) в Look. К счастью, так как мы были осторожны и использовали GetName, GetNum, и Match для большей части обработки входного потока, только эти три процедуры (плюс Init) необходимо изменить.

Неудивительно, что мы начинаем с еще одной подпрограммы распознавания:

```
{--------------------------------------------------------------}
{ Recognize White Space }
function IsWhite(c: char): boolean;
begin
IsWhite := c in [' ', TAB];
end;
{--------------------------------------------------------------}
```

Нам также нужна процедура, «съедающая» символы пробела до тех пор, пока не найдет отличный от пробела символ:

```
{--------------------------------------------------------------}
{ Skip Over Leading White Space }
procedure SkipWhite;
begin
while IsWhite(Look) do
GetChar;
end;
{--------------------------------------------------------------}
```

Сейчас добавьте вызовы SkipWhite в Match, GetName и GetNum как показано ниже:

```
{--------------------------------------------------------------}
{ Match a Specific Input Character }
procedure Match(x: char);
begin
if Look &amp;lt;&amp;gt; x then Expected('''' + x + '''')
else begin
GetChar;
SkipWhite;
end;
```

```
end;
{--------------------------------------------------------------}
{ Get an Identifier }
function GetName: string;
var Token: string;
begin
Token := '';
if not IsAlpha(Look) then Expected('Name');
while IsAlNum(Look) do begin
Token := Token + UpCase(Look);
GetChar;
end;
GetName := Token;
SkipWhite;
end;
{--------------------------------------------------------------}
{ Get a Number }
function GetNum: string;
var Value: string;
begin
Value := '';
if not IsDigit(Look) then Expected('Integer');
while IsDigit(Look) do begin
Value := Value + Look;
GetChar;
end;
GetNum := Value;
SkipWhite;
end;
{--------------------------------------------------------------}
```

(Обратите внимание, как я немного реорганизовал Match без изменения функциональности.)

Наконец, мы должны пропустить начальные пробелы в том месте, где мы «запускаем помпу» в Init:

```
{--------------------------------------------------------------}
{ Initialize }
procedure Init;
begin
GetChar;
SkipWhite;
end;
{--------------------------------------------------------------}
```

Внесите эти изменения и повторно откомпилируйте программу. Вы обнаружите, что необходимо переместить Match ниже SkipWhite чтобы избежать сообщение об ошибке от компилятора Pascal. Протестируйте программу как всегда, чтобы удостовериться, что она работает правильно.

Поскольку мы сделали довольно много изменений в течение этого урока, ниже я воспроизвожу полный текст синтаксического анализатора:

```
{--------------------------------------------------------------}
program parse;
{--------------------------------------------------------------}
{ Constant Declarations }
```

```
const TAB = ^I;
CR = ^M;
_
 Variable Declarations
var Look: char;  Lookahead Character
_
 Read New Character From Input Stream
procedure GetChar;
begin
Read(Look);
end;
_
 Report an Error
procedure Error(s: string);
begin
WriteLn;
WriteLn(^G, 'Error: ', s, '.');
end;
_
 Report Error and Halt
procedure Abort(s: string);
begin
Error(s);
Halt;
end;
_
 Report What Was Expected
procedure Expected(s: string);
begin
Abort(s + ' Expected');
end;
_
 Recognize an Alpha Character
function IsAlpha(c: char): boolean;
begin
IsAlpha := UpCase(c) in ['A'..'Z'];
end;
_
 Recognize a Decimal Digit
function IsDigit(c: char): boolean;
begin
IsDigit := c in ['0'..'9'];
end;
_
 Recognize an Alphanumeric
function IsAlNum(c: char): boolean;
begin
IsAlNum := IsAlpha(c) or IsDigit(c);
end;
_
 Recognize an Addop
function IsAddop(c: char): boolean;
begin
IsAddop := c in ['+', '-'];
end;
_
 Recognize White Space
```

```
function IsWhite(c: char): boolean;
begin
IsWhite := c in [' ', TAB];
end;
_
 Skip Over Leading White Space
procedure SkipWhite;
begin
while IsWhite(Look) do
GetChar;
end;
_
 Match a Specific Input Character
procedure Match(x: char);
begin
if Look &lt;&gt; x then Expected("" + x + "")
else begin
GetChar;
SkipWhite;
end;
end;
_
 Get an Identifier
function GetName: string;
var Token: string;
begin
Token := '';
if not IsAlpha(Look) then Expected('Name');
while IsAlNum(Look) do begin
Token := Token + UpCase(Look);
GetChar;
end;
GetName := Token;
SkipWhite;
end;
_
 Get a Number
function GetNum: string;
var Value: string;
begin
Value := '';
if not IsDigit(Look) then Expected('Integer');
while IsDigit(Look) do begin
Value := Value + Look;
GetChar;
end;
GetNum := Value;
SkipWhite;
end;
_
 Output a String with Tab
procedure Emit(s: string);
begin
Write(TAB, s);
end;
_
 Output a String with Tab and CRLF
```

```
procedure EmitLn(s: string);
begin
Emit(s);
WriteLn;
end;
{--------------------------------------------------------------}
{ Parse and Translate a Identifier }
procedure Ident;
var Name: string[8];
begin
Name:= GetName;
if Look = '(' then begin
Match('(');
Match(')');
EmitLn('BSR ' + Name);
end
else
EmitLn('MOVE ' + Name + '(PC),D0');
end;
{--------------------------------------------------------------}
{ Parse and Translate a Math Factor }
procedure Expression; Forward;
procedure Factor;
begin
if Look = '(' then begin
Match('(');
Expression;
Match(')');
end
else if IsAlpha(Look) then
Ident
else
EmitLn('MOVE #' + GetNum + ',D0');
end;
{--------------------------------------------------------------}
{ Recognize and Translate a Multiply }
procedure Multiply;
begin
Match('*');
Factor;
EmitLn('MULS (SP)+,D0');
end;
{--------------------------------------------------------------}
{ Recognize and Translate a Divide }
procedure Divide;
begin
Match('/');
Factor;
EmitLn('MOVE (SP)+,D1');
EmitLn('EXS.L D0');
EmitLn('DIVS D1,D0');
end;
{--------------------------------------------------------------}
{ Parse and Translate a Math Term }
procedure Term;
begin
Factor;
```

```
while Look in ['*', '/'] do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'*': Multiply;
'/': Divide;
end;
end;
end;
_
 Recognize and Translate an Add
procedure Add;
begin
Match('+');
Term;
EmitLn('ADD (SP)+,D0');
end;
_
 Recognize and Translate a Subtract
procedure Subtract;
begin
Match('-');
Term;
EmitLn('SUB (SP)+,D0');
EmitLn('NEG D0');
end;
_
 Parse and Translate an Expression
procedure Expression;
begin
if IsAddop(Look) then
EmitLn('CLR D0')
else
Term;
while IsAddop(Look) do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'+': Add;
'-': Subtract;
end;
end;
end;
_
 Parse and Translate an Assignment Statement
procedure Assignment;
var Name: string[8];
begin
Name := GetName;
Match('=');
Expression;
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE D0,(A0)')
end;
_
 Initialize
procedure Init;
begin
GetChar;
```

```
SkipWhite;
end;
–
 Main Program
begin
Init;
Assignment;
If Look &lt;&gt; CR then Expected('NewLine');
end.
–
```

Теперь синтаксический анализатор закончен. Он получил все возможности, которые мы можем разместить в однострочном «компиляторе». Сохраните его в безопасном месте. В следующий раз мы перейдем к новой теме, но мы все рано будем некоторое время говорить о выражениях. В следующей главе я планирую рассказать немного об интерпретаторах в противоположность компиляторам и показать вам как немного изменяется структура синтаксического анализатора в зависимости от изменения характера принимаемых действий. Информация, которую мы рассмотрим, хорошо послужит нам позднее, даже если вы не интересуетесь интерпретаторами. Увидимся в следующий раз.

Интерпретаторы

ВВЕДЕНИЕ

В трех первых частях этой серии мы рассмотрели синтаксический анализ и компиляцию математических выражений, постепенно и методично пройдя от очень простых односимвольных «выражений», состоящих из одного терма, через выражения в более общей форме и закончив достаточно полным синтаксическим анализатором, способным анализировать и транслировать операции присваивания с многосимвольными токенами, вложенными пробелами и вызовами функций. Сейчас я собираюсь провести вас сквозь этот процесс еще раз, но уже с целью интерпретации а не компиляции объектного кода.

Если эта серия о компиляторах, то почему мы должны беспокоиться об интерпретаторах? Просто я хочу чтобы вы увидели как изменяется характер синтаксического анализатора при изменении целей. Я также хочу объединить понятия этих двух типов трансляторов, чтобы вы могли видеть не только различия но и сходства.

Рассмотрим следующее присваивание:

x = 2 * y + 3

В компиляторе мы хотим заставить центральный процессор выполнить это присваивание во время выполнения. Сам транслятор не выполняет никаких арифметических операций… он только выдает объектный код, который заставит процессор сделать это когда код выполнится. В примере выше компилятор выдал бы код для вычисления значения выражения и сохранения результата в переменной x.

Для интерпретатора, напротив, никакого объектного кода не генерируется. Вместо этого арифметические операции выполняются немедленно как только происходит синтаксический анализ. К примеру, когда синтаксический анализ присваивания завершен, x будет содержать новое значение.

Метод, который мы применяем во всей этой серии, называется «синтаксически-управляемым переводом». Как вы знаете к настоящему времен, структура синтаксического анализатора очень близко привязана к синтаксису анализируемых нами конструкций. Мы создали процедуры на Pascal, которые распознают каждую конструкцию языка. Каждая из этих конструкций (и процедур) связана с соответствующим «действием», которое выполняет все необходимое как только конструкция распознана. В нашем компиляторе каждое действие

включает выдачу объектного кода для выполнения позднее во время исполнения. В интерпретаторе каждое действие включает что-то для немедленного выполнения.

Что я хотел бы, чтобы вы увидели, это то, что план… структура… синтаксического анализатора не меняется. Изменяются только действия. Так что, если вы можете написать интерпретатор для данного языка, то вы можете также написать и компилятор, и наоборот. Однако, как вы увидите, имеются и отличия, и значительные. Поскольку действия различны, процедуры, завершающие распознавание, пишутся по-разному. Характерно, что в интерпретаторе завершающие подпрограммы распознавания написаны как функции, возвращающие числовое значение вызвавшей их программе. Ни одна из подпрограмм анализа нашего компилятора не делает этого.

Наш компилятор, фактически, это то, что мы могли бы назвать «чистым» компилятором. Как только конструкция распознана, объектный код выдается немедленно. (Это одна из причин, по которым код не очень эффективный.) Интерпретатор, который мы собираемся построить, является чистым интерпретаторов в том смысле, что здесь нет никакой трансляции типа «токенизации», выполняемой над исходным текстом. Это две крайности трансляции. В реальном мире трансляторы не являются такими чистыми, но стремятся использовать часть каждой методики.

Я могу привести несколько примеров. Я уже упомянул один: большинство интерпретаторов, типа Microsoft BASIC, к примеру, транслируют исходный текст (токенизируют его) в промежуточную форму, чтобы было легче выполнять синтаксический анализ в реальном режиме времени.

Другой пример – ассемблер. Целью ассемблера, конечно, является получение объектного кода и он обычно выполняет это по однозначному принципу: одна инструкция на строку исходного кода. Но почти все ассемблеры также разрешают использовать выражения как параметры. В этом случае выражения всегда являются константами, и ассемблер не предназначен выдавать для них объектный код. Скорее он «интерпретирует» выражение и вычисляет соответствующее значение, которое фактически и выдается с объектным кодом.

Фактически, мы могли бы использовать часть этого сами. Транслятор, который мы создали в предыдущей главе, будет покорно выплевывать объектный код для сложных выражений, даже если каждый терм в выражении будет константой. В этом случае было бы гораздо лучше, если бы транслятор вел себя немного как интерпретатор и просто вычислял соответствующее значение константы.

В теории компиляции существует понятие, называемое «ленивой» трансляцией. Идея состоит в том, что вы не просто выдаете код при каждом действии. Фактически, в крайнем случае вы не выдаете что-либо вообще до тех пор, пока это не будет абсолютно необходимо. Для выполнения этого, действия, связанные с подпрограммами анализа, обычно не просто выдают код. Иногда они это делают, но часто они просто возвращают информацию обратно вызвавшей программе. Вооружившись этой информацией, вызывающая программа может затем сделать лучший выбор того, что делать.

К примеру, для данного выражения

x = x + 3 – 2 – (5 – 4)

наш компилятор будет покорно выплевывать поток из 18 инструкций для загрузки каждого параметра в регистры, выполнения арифметических действий и сохранения результата. Ленивая оценка распознала бы, что выражение, содержащее константы, может быть рассчитано во время компиляции и уменьшила бы выражение до

x = x + 0

Даже ленивая оценка была бы затем достаточно умной, чтобы понять, что это эквивалентно

x = x,

что совсем не требует никаких действий. Мы смогли уменьшить 18 инструкций до нуля!

Обратите внимание, что нет никакой возможности оптимизировать таким способом наш компилятор, потому что каждое действие выполняется в нем немедленно.

Ленивая оценка выражений может произвести значительно лучший объектный код чем тот который мы могли произвести. Я, тем не менее, предупреждаю вас: это значительно усложняет код синтаксического анализатора, потому что каждая подпрограмма теперь должна принять решение относительно того, выдать объектный код или нет. Ленивая оценка конечно же названа так не потому, что она проще для создателей компиляторов!

Так как мы действуем в основном по принципу KISS, я не буду более углубляться в эту тему. Я только хочу, чтобы вы знали, что вы можете получить некоторую оптимизацию кода, объединяя методы компиляции и интерпретации. В частности Вы должны знать, что подпрограммы синтаксического анализа в более интеллектуальном трансляторе обычно что-то возвращают вызвавшей их программе и иногда сами ожидают этого. Эта главная причина обсуждения интерпретаторов в этой главе.

ИНТЕРПРЕТАТОР

Итак, теперь, когда вы знаете почему мы принялись за все это, давайте начнем. Просто для того, чтобы дать вам практику, мы начнем с пустого Cradle и создадим транслятор заново. На этот раз, конечно, мы сможем двигаться немного быстрее.

Так как сейчас мы собираемся выполнять арифметические действия, то первое, что мы должны сделать – изменить функцию GetNum, которая до настоящего момента всегда возвращала символ (или строку). Лучше если сейчас она будет возвращать целое число. Сделайте копию Cradle (на всякий случай не изменяйте сам Cradle!!) и модифицируйте GetNum следующим образом:

```
{--------------------------------------------------------------}
{ Get a Number }
function GetNum: integer;
begin
if not IsDigit(Look) then Expected('Integer');
GetNum := Ord(Look) - Ord('0');
GetChar;
end;
{--------------------------------------------------------------}
```

Затем напишите следующую версию Expression:

```
{--------------------------------------------------------------}
{ Parse and Translate an Expression }
function Expression: integer;
begin
Expression := GetNum;
end;
{--------------------------------------------------------------}
```

И, наконец, вставьте

Writeln(Expression);

в конец основной программы. Теперь откомпилируйте и протестируйте.

Все, что эта программа делает – это «анализ» и трансляция «выражения», состоящего из одиночного целого числа. Как обычно, вы должны удостовериться, что она обрабатывает числа от 0 до 9 и выдает сообщение об ошибке для чего-либо другого. Это не должно занять у вас много времени!

Теперь давайте расширим ее, включив поддержку операций сложения. Измените Expression так:

```
_
 Parse and Translate an Expression
function Expression: integer;
var Value: integer;
begin
if IsAddop(Look) then
Value := 0
else
Value := GetNum;
while IsAddop(Look) do begin
case Look of
'+': begin
Match('+');
Value := Value + GetNum;
end;
'-': begin
Match('-');
Value := Value – GetNum;
end;
end;
end;
Expression := Value;
end;
_
```

Структура Expression, конечно, схожа с тем, что мы делали ранее, так что мы не будем иметь слишком много проблем при ее отладке. Тем не менее это была серьезная разработка, не так ли? Процедуры Add и Subtract исчезли! Причина в том, что для выполнения необходимых действий нужны оба аргумента операции. Я мог бы сохранить эти процедуры и передавать в них значение выражения на данный момент, содержащееся в Value. Но мне показалось более правильным оставить Value как строго локальную переменную, что означает, что код для Add и Subtract должен быть помещен вместе. Этот результат наводит на мысль, что хотя разработанная нами структура была хорошей и проверенной для нашей бесхитростной схемы трансляции, она возможно не могла бы использоваться с ленивой оценкой. Эту небольшую интересную новость нам возможно необходимо иметь в виду в будущем.

Итак, транслятор работает? Тогда давайте сделаем следующий шаг. Несложно понять, что процедура Term должна выглядеть также. Замените каждый вызов GetNum в функции Expression на вызов Term и затем наберите следующую версию Term:

```
_
 Parse and Translate a Math Term
function Term: integer;
var Value: integer;
begin
Value := GetNum;
while Look in ['*', '/'] do begin
case Look of
'*': begin
Match('*');
Value := Value * GetNum;
end;
'/': begin
Match('/');
Value := Value div GetNum;
```

```
end;
end;
end;
Term := Value;
end;
–
```

Теперь испробуйте. Не забудьте двух вещей: во-первых мы имеем дело с целочисленным делением, поэтому, например, 1/3 выдаст ноль. Во-вторых, даже если мы можем получать на выходе многозначные числа, входные числа все еще ограничены одиночной цифрой.

Сейчас это выглядит как глупое ограничение, так как мы уже видели как легко может быть расширена функция GetNum. Так что давайте исправим ее прямо сейчас. Вот новая версия:

```
–
 Get a Number
function GetNum: integer;
var Value: integer;
begin
Value := 0;
if not IsDigit(Look) then Expected('Integer');
while IsDigit(Look) do begin
Value := 10 * Value + Ord(Look) – Ord('0');
GetChar;
end;
GetNum := Value;
end;
–
```

Если вы откомпилировали и протестировали эту версию интерпретатора, следующим шагом должна быть установка функции Factor, поддерживающей выражения в скобках. Мы задержимся немного дольше на именах переменных. Сначала измените ссылку на GetNum в функции Term, чтобы вместо нее вызывалась функция Factor. Теперь наберите следующую версию Factor:

```
–
 Parse and Translate a Math Factor
function Expression: integer; Forward;
function Factor: integer;
begin
if Look = '(' then begin
Match('(');
Factor := Expression;
Match(')');
end
else
Factor := GetNum;
end;
–
```

Это было довольно легко, а? Мы быстро пришли к полезному интерпретатору.

НЕМНОГО ФИЛОСОФИИ

Прежде чем двинуться дальше, я бы хотел обратить ваше внимание на кое-что. Я говорю о концепции, которую мы использовали на всех этих уроках, но которую я явно не упомянул до сих пор. Я думаю, что пришло время сделать это, так как эта концепция настолько полезная и настолько мощная, что она стирает все различия между тривиально простым синтаксическим анализатором и тем, который слишком сложен для того, чтобы иметь с ним дело.

В ранние дни технологии компиляции люди тратили ужасно много времени на выяснение того, как работать с такими вещами как приоритет операторов… способа, который определяет приоритет операторов умножения и деления над сложением и вычитанием и т.п. Я помню одного своего коллегу лет тридцать назад и как возбужденно он выяснял как это делается. Используемый им метод предусматривал создание двух стеков, в которые вы помещали оператор или операнд. С каждым оператором был связан уровень приоритета и правила требовали, чтобы вы фактически выполняли операцию («уменьшающую» стек) если уровень приоритета на вершине стека был корректным. Чтобы сделать жизнь более интересной оператор типа ")" имел различные приоритеты в зависимости от того, был он уже в стеке или нет. Вы должны были дать ему одно значение перед тем как поместите в стек и другое, когда решите извлечь из стека. Просто для эксперимента я самостоятельно поработал со всем этим несколько лет назад и могу сказать вам, что это очень сложно.

Мы не делали что-либо подобное. Фактически, к настоящему времени синтаксический анализ арифметических выражений должен походить на детскую игру. Как мы оказались настолько удачными? И куда делся стек приоритетов?

Подобная вещь происходит в нашем интерпретаторе выше. Вы просто знаете, что для того, чтобы выполнить вычисления арифметических выражений (в противоположность их анализу), должны иметься числа, помещенные в стек. Но где стек?

Наконец, в учебниках по компиляторам имеются разделы, где обсуждены стеки и другие структуры. В другом передовом методе синтаксического анализа (LR) используется явный стек. Фактически этот метод очень похож на старый способ вычисления арифметических выражений. Другая концепция – это синтаксическое дерево. Авторы любят рисовать диаграммы из токенов в выражении объединенные в дерево с операторами во внутренних узлах. И снова, где в нашем методе деревья и стеки? Мы не видели ничего такого. Во всех случаях ответ в том, что эти структуры не явные а неявные. В любом машинном языке имеется стек, используемый каждый раз, когда вы вызываете подпрограмму. Каждый раз, когда вызывается подпрограмма, адрес возврата помещается в стек ЦПУ. В конце подпрограммы адрес выталкивается из стека и управление передается на этот адрес. В рекурсивном языке, таком как Pascal, могут также иметься локальные данные, помещенные в стек, и они также возвращаются когда это необходимо.

Например функция Expression содержит локальный параметр, названный Value, которому присваивается значение при вызове Term. Предположим, при следующем вызове Term для второго аргумента, что Term вызывает Factor, который рекурсивно вызывает Expression снова. Этот «экземпляр» Expression получает другое значение для его копии Value. Что случится с первым значением Value? Ответ: он все еще в стеке и будет здесь снова, когда мы возвратимся из нашей последовательности вызовов.

Другими словами, причина, по которой это выглядит так просто в том, что мы максимально использовали ресурсы языка. Уровни иерархии и синтаксические деревья присутствуют здесь, все правильно, но они скрыты внутри структуры синтаксического анализатора и о них заботится порядок в котором вызываются различные процедуры. Теперь, когда вы увидели, как мы делаем это, возможно трудно будет придумать как сделать это каким-либо другим способом. Но я могу сказать вам, что это заняло много лет для создателей компиляторов. Первые компиляторы были слишком сложными. Забавно, как работа становится легче с небольшой практикой.

Вывод из всего того, что я привел здесь, служит и уроком и предупреждением. Урок:

дела могут быть простыми если вы приметесь за них с правильной стороны. Предупреждение: смотрите, что делаете. Если вы делаете что-либо самостоятельно и начинаете испытывать потребность в отдельном стеке или дереве, возможно это время спросить себя, правильно ли вы смотрите на вещи. Возможно вы просто не используете возможностей языка так как могли бы.

Следующий шаг – добавление имен переменных. Сейчас, однако, мы имеем небольшую проблему. В случае с компилятором мы не имели проблем при работе с именами переменных… мы просто выдавали эти имена ассемблеру и позволяли остальной части программы заботиться о распределении для них памяти. Здесь же, напротив, у нас должна быть возможность извлекать значения переменных и возвращать их как значение функции Factor. Нам необходим механизм хранения этих переменных.

В ранние дни персональных компьютеров существовал Tiny Basic. Он имел в общей сложности 26 возможных переменных: одна на каждую букву алфавита. Это хорошо соответствует нашей концепции односимвольных токенов, так что мы испробуем этот же прием. В начале интерпретатора, сразу после объявления переменной Look, вставьте строку:

Table: Array['A'..'Z'] of integer;

Мы также должны инициализировать массив, поэтому добавьте следующую процедуру:

```
{--------------------------------------------------------------}
{ Initialize the Variable Area }
procedure InitTable;
var i: char;
begin
for i := 'A' to 'Z' do
Table[i] := 0;
end;
{--------------------------------------------------------------}
```

Вы также должны вставить вызов InitTable в процедуру Init. Не забудьте сделать это, иначе результат может удивить вас!

Теперь, когда у нас есть массив переменных, мы можем модифицировать Factor так, чтобы он их использовал. Так как мы не имеем (пока) способа для установки значения переменной, Factor будет всегда возвращать для них нулевые значения, но давайте двинемся дальше и расширим его. Вот новая версия:

```
{--------------------------------------------------------------}
{ Parse and Translate a Math Factor }
function Expression: integer; Forward;
function Factor: integer;
begin
if Look = '(' then begin
Match('(');
Factor := Expression;
Match(')');
end
else if IsAlpha(Look) then
Factor := Table[GetName]
else
Factor := GetNum;
end;
{--------------------------------------------------------------}
```

Как всегда откомпилируйте и протестируйте эту версию программы Даже притом, что

все переменные сейчас равны нулю, по крайней мере мы можем правильно анализировать законченные выражения, так же как и отлавливать любые неправильно оформленные.

Я предполагаю вы уже знаете следующий шаг: мы должны добавить операции присваивания, чтобы мы могли помещать что-нибудь в переменные. Сейчас давайте будем «однострочниками», хотя скоро мы сможем обрабатывать множество операторов.

Операция присваивания похожа на то, что мы делали раньше:

```
{--------------------------------------------------------------}
{ Parse and Translate an Assignment Statement }
procedure Assignment;
var Name: char;
begin
Name := GetName;
Match('=');
Table[Name] := Expression;
end;
{--------------------------------------------------------------}
```

Чтобы протестировать ее, я добавил временный оператор write в основную программу для вывода значения A. Затем я протестировал ее с различными присваиваниями.

Конечно, интерпретируемый язык, который может воспринимать только одну строку программы не имеет большой ценности. Поэтому нам нужно обрабатывать множество утверждений. Это просто означает что необходимо поместить цикл вокруг вызова Assignment. Давайте сделаем это сейчас. Но что должно быть критерием выхода из цикла? Рад, что вы спросили, потому что это поднимает вопрос, который мы были способны игнорировать до сих пор.

Одной из наиболее сложных вещей в любом трансляторе является определение момента когда необходимо выйти из данной конструкции и продолжить выполнение. Пока это не было для нас проблемой, потому что мы допускали только одну конструкцию… или выражение или операцию присваивания. Когда мы начинаем добавлять циклы и различные виды операторов, вы найдете, что мы должны быть очень осторожны, чтобы они завершались правильно. Если мы помещаем наш интерпретатор в цикл, то нам нужен способ для выхода из него. В прерывании по концу строки нет ничего хорошего, поскольку с его помощью мы переходим к следующей строке. Мы всегда могли позволить нераспознаваемым символам прерывать выполнение, но это приводило бы к завершению каждой программы сообщением об ошибке, что конечно выглядит несерьезно.

Нам нужен завершающий символ. Я выступаю за завершающую точку в Pascal ("."). Небольшое осложнение состоит в том, что Turbo Pascal завершает каждую нормальную строку двумя символами: возврат каретки (CR) и перевод строки (LF). В конце каждой строки мы должны «съедать» эти символы перед обработкой следующей. Естественным способом было бы сделать это в процедуре Match за исключением того, что сообщение об ошибке Match выводит ожидаемые символы, что для CR и LF не будет выглядеть так хорошо. Для этого нам нужна специальная процедура, которую мы, без сомнения, будем использовать много раз. Вот она:

```
{--------------------------------------------------------------}
{ Recognize and Skip Over a Newline }
procedure NewLine;
begin
if Look = CR then begin
GetChar;
if Look = LF then
GetChar;
end;
```

```
end;
–
```

Вставьте эту процедуру в любом удобном месте… я поместил ее сразу после Match. Теперь перепишите основную программу, чтобы она выглядела следующим образом:

```
–
 Main Program
begin
Init;
repeat
Assignment;
NewLine;
until Look = '.';
end.
–
```

Обратите внимание, что проверка на CR теперь исчезла и что также нет проверки на ошибку непосредственно внутри NewLine. Это нормально… все оставшиеся фиктивные символы будут отловлены в начале следующей операции присваивания.

Хорошо, сейчас мы имеем функционирующий интерпретатор. Однако, это не дает нам много хорошего, так как у нас нет никакого способа для ввода или вывода данных. Уверен что нам помогут несколько подпрограмм ввода/вывода!

Тогда давайте завершим этот урок добавив подпрограммы ввода/вывода. Так как мы придерживаемся односимвольных токенов, я буду использовать знак "?" для замены операции чтения, знак "!" для операции записи и символ, немедленно следующий после них, который будет использоваться как односимвольный «список параметров». Вот эти подпрограммы:

```
–
 Input Routine
procedure Input;
begin
Match('?');
Read(Table[GetName]);
end;
–
 Output Routine
procedure Output;
begin
Match('!');
WriteLn(Table[GetName]);
end;
–
```

Я полагаю они не очень причудливы… например нет никакого символа приглашения при вводе… но они делают свою работу.

Соответствующие изменения в основной программе показаны ниже. Обратите внимание, что мы используем обычный прием – оператор выбора по текущему предсказывающему символу, чтобы решить что делать.

```
–
 Main Program
begin
Init;
```

```
repeat
case Look of
'?': Input;
'!': Output;
else Assignment;
end;
NewLine;
until Look = '.';
end.
–
```

Теперь вы закончили создание настоящего, работающего интерпретатора. Он довольно скудный, но работает совсем как «большой мальчик». Он включает три вида операторов (и может различить их!), 26 переменных и операторы ввода/вывода. Единственное, в чем он действительно испытывает недостаток – это операторы управления, подпрограммы и некоторые виды функций для редактирования программы. Функции редактирования программ я собираюсь пропустить. В конце концов, мы здесь не для того, чтобы создать готовый продукт, а чтобы учиться. Управляющие операторы мы раскроем в следующей главе, а подпрограммы вскоре после нее. Я стремлюсь продолжать дальше, поэтому мы оставим интерпретатор в его текущем состоянии.

Я надеюсь, к настоящему времени вы убедились, что ограничение имен одним символом и обработка пробелов это вещи о которых легко позаботиться, как мы сделали это на последнем уроке. На этот раз, если вам захотелось поиграть с этими расширениями, будьте моим гостем… они «оставлены как упражнение для студента». Увидимся в следующий раз.

Управляющие конструкции

ВВЕДЕНИЕ

В четырех первых главах этой серии мы сконцентрировали свое внимание на синтаксическом анализе математических выражений и операций присваивания. В этой главе мы остановимся на новой и захватывающей теме: синтаксическом анализе и трансляции управляющих конструкций таких как, например, операторы IF.

Эта тема дорога для моего сердца, потому что является для меня поворотной точкой. Я играл с синтаксическим анализом выражений также как мы делали это в этой серии, но я все же чувствовал, что нахожусь еще очень далеко от возможности поддержки полного языка. В конце концов, реальные языки имеют ветвления, циклы, подпрограммы и все такое. Возможно вы разделяли некоторые из этих мыслей. Некоторое время назад, тем не менее, я должен был реализовать управляющие конструкции для структурного препроцессора ассемблера, который я писал. Вообразите мое удивление, когда я обнаружил, что это было гораздо проще, чем синтаксический анализ выражений, через который я уже прошел. Я помню подумал «Эй, это же просто!». После того, как мы закончим этот урок, я готов поспорить, что вы будете думать так же.

ПЛАН

Далее мы снова начнем с пустого Cradle и, как мы делали уже дважды до этого, будем строить программу последовательно. Мы также сохраним концепцию односимвольных токенов, которая так хорошо служила нам до настоящего времени. Это означает, что «код» будет выглядеть немного забавным с "i" вместо IF, "w" вместо WHILE и т.д. Но это поможет нам узнать основные понятия не беспокоясь о лексическом анализе. Не бойтесь... в конечном

счете мы увидим что-то похожее на «настоящий» код.

Я также не хочу, чтобы мы увязли в работе с какими либо операторами кроме ветвлений, такими как операции присваивания, с которыми мы уже работали. Мы уже показали, что можем обрабатывать их, так что нет никакого смысла таскать этот лишний багаж в течение предстоящих занятий. Вместо этого я буду использовать анонимный оператор «other» для замены неуправляющих операторов. Мы должны генерировать для них некоторый обьектный код (мы возвращаемся к компиляции а не интерпретации), так что за неимением чего-либо другого я буду просто повторять входной символ.

Итак, тогда, начав с еще одной копии Cradle, давайте определим процедуру:

```
–
 Recognize and Translate an «Other»
procedure Other;
begin
EmitLn(GetName);
end;
–
```

Теперь включим ее вызов в основную программу таким образом:

```
–
 Main Program
begin
Init;
Other;
end.
–
```

Запустите программу и посмотрите, что вы получили. Не очень захватывающе, не так ли? Но не зацикливайтесь на этом, это только начало, результат будет лучше.

Первое, что нам нужно – это возможность работать с более чем одним оператором, так как однострочные ветвления довольно ограничены. Мы делали это на последнем занятии по интерпретации, но сейчас давайте будем немного более формальными. Рассмотрите следующую БНФ:

<program> ::= <block> END

<block> ::= [<statement>]*

Это означает, что программа определена как блок, завершаемый утверждением END. Блок, в свою очередь, состоит из нуля или более операторов. Пока у нас есть только один вид операторов.

Что является признаком окончания блока? Это просто любая конструкция, не являющаяся оператором «other». Сейчас это только утверждение END.

Вооружившись этими идеями, мы можем приступать к созданию нашего синтаксического анализатора. Код для program (мы должны назвать его DoProgram, иначе Pascal будет ругаться) следующий:

```
–
 Parse and Translate a Program
procedure DoProgram;
begin
Block;
if Look &lt;&gt; 'e' then Expected('End');
EmitLn('END')
end;
–
```

Обратите внимание, что я выдаю ассемблеру команду «END», что своего рода расставляет знаки препинания в выходном коде и заставляет чувствовать, что мы анализируем здесь законченную программу.

Код для Block:

```
_
 Recognize and Translate a Statement Block
procedure Block;
begin
while not(Look in ['e']) do begin
Other;
end;
end;
_
```

(Из формы процедуры вы видите, что мы собираемся постепенно ее расширять!)

ОК, вставьте эти подпрограммы в вашу программу. Замените вызов Block в основной программе на вызов DoProgram. Теперь испытайте ее и посмотрите как она работает. Хорошо, все еще не так много, но мы становимся все ближе.

НЕМНОГО ОСНОВ

Прежде чем мы начнем определять различные управляющие конструкции, мы должны положить немного более прочное основание. Во-первых, предупреждаю: я не буду использовать для этих конструкций тот же самый синтаксис с которым вы знакомы по Паскалю или Си. К примеру синтаксис Паскаль для IF такой:

IF <condition> THEN <statement>

(где <statement>, конечно, может быть составным.)

Синтаксис С аналогичен этому:

IF (<condition>) <statement>

Вместо этого я буду использовать нечто более похожее на Ada:

IF <condition> <block> ENDIF

Другими словами, конструкция IF имеет специфический символ завершения. Это позволит избежать висячих else Паскаля и Си и также предотвращает необходимость использовать скобки или begin-end. Синтаксис, который я вам здесь показываю, фактически является синтаксисом языка KISS, который я буду детализировать в следующих главах. Другие конструкции также будут немного отличаться. Это не должно быть для вас большой проблемой. Как только вы увидите, как это делается, вы поймете, что в действительности не имеет большого значения, какой конкретный синтаксис используется. Как только синтаксис определен, включить его в код достаточно просто.

Теперь, все конструкции, с которыми мы будем иметь дело, включают передачу управления, что на уровне ассемблера означает условные и/или безусловные переходы. К примеру простой оператор IF:

IF <condition> A ENDIF B...

должен быть переведен в:

Если условие не выполнено то переход на L

A

L: B

...

Ясно, что нам понадобятся несколько процедур, которые помогут нам работать с этими переходами. Ниже я определил две из них. Процедура NewLabel генерирует уникальные метки. Это сделано с помощью простого способа называть каждую метку 'Lnn', где nn – это

номер метки, начинающийся с нуля. Процедура PostLabel просто выводит метки в соответствующем месте.

Вот эти две подпрограммы:

```
{
 Generate a Unique Label
}
function NewLabel: string;
var S: string;
begin
Str(LCount, S);
NewLabel := 'L' + S;
Inc(LCount);
end;
{
 Post a Label To Output
}
procedure PostLabel(L: string);
begin
WriteLn(L, ':');
end;
{
```

Заметьте, что мы добавили новую глобальную переменную LCount, так что вы должны изменить раздел описания переменных в начале программы, следующим образом:

```
var Look : char;  Lookahead Character
Lcount: integer;  Label Counter
```

Также добавьте следующий дополнительный инициализирующий код в Init:

```
LCount := 0;
```

(Не забудьте сделать это, иначе ваши метки будут выглядеть действительно странными!).

В этом месте я также хотел бы показать вам новый вид нотации. Если вы сравните форму оператора IF, указанную выше, с ассемблерным кодом, который должен быть получен, то вы можете увидеть, что существуют некоторые определенные действия, связанные с каждым ключевым словом в операторе:

IF: Сначала получить условие и выдать код для него. Затем создать уникальную метку и выдать переход если условие ложно.

ENDIF: Выдать метку.

Эти действия могут быть показаны очень кратко, если мы запишем синтаксис таким образом:

IF
<condition> Condition;
L = NewLabel;
Emit(Branch False to L);
<block>
ENDIF PostLabel(L)

Это пример синтаксически-управляемого перевода. Мы уже делали все это... мы просто никогда прежде не записывали это таким образом. Содержимое фигурных скобок представляет собой действия, которые будут выполняться. Хорошо в этом способе представления то, что он не только показывает что мы должны распознать, но также и действия, которые мы должны выполнить и в каком порядке. Как только мы получаем такой синтаксис, код возникает почти сам собой.

Почти единственное, что осталось сделать – конкретизировать то, что мы подразумеваем под «Переход если условие ложно».

Я полагаю, что должен быть код, выполняющийся для <condition>, который будет выполнять булеву алгебру и вычислять некоторый результат. Он также должен установить флажки условий, соответствующие этому результату. Теперь, обычным соглашением для булевых переменных является использование 0000 для представления значения «ложь» и какого-либо другого значения (кто-то использует FFFF, кто-то 0001) для представления «истины».

Процессор 68000 устанавливает флажки условий всякий раз, когда любые данные перемещаются или рассчитываются. Если данные равны 0000 (что соответствует условию ложь, запомните) будет установлен флажок ноль. Код для «перехода по нулю» – BEQ. Таким образом

BEQ <=> Переход если ложь

BNE <=> Переход если истина

По природе вещей большинство ветвлений, которые мы увидим, будут BEQ... мы будем обходить вокруг кода, который должен выполняться когда условие истинно.

ОПЕРАТОР IF

После этого небольшого пояснения метода мы наконец готовы начать программирование синтаксического анализатора для условного оператора. Фактически, мы уже почти сделали это! Как обычно я буду использовать наш односимвольный подход, с символом "i" вместо «IF» и "e" вместо «ENDIF» (также как и END... это двойственная природа не вызывает никакого беспорядка). Я также пока полностью пропущу символ для условия ветвления, который мы все еще должны определить.

Код для DoIf:

```
_
 Recognize and Translate an IF Construct
procedure Block; Forward;
procedure DoIf;
var L: string;
begin
Match('i');
L := NewLabel;
Condition;
EmitLn('BEQ ' + L);
Block;
Match('e');
PostLabel(L);
end;
_
```

Добавьте эту подпрограмму в вашу программу и измените Block так, чтобы он ссылался на нее как показано ниже:

```
_
 Recognize and Translate a Statement Block
procedure Block;
begin
while not(Look in ['e']) do begin
case Look of
'i': DoIf;
```

```
'o': Other;
end;
end;
end;
_
```

Обратите внимание на обращение к процедуре Condition. В конечном итоге мы напишем подпрограмму, которая сможет анализировать и транслировать любое логическое условие которое мы ей дадим. Но это уже тема для отдельной главы (фактически следующей). А сейчас давайте просто заменим ее макетом, который выдает некоторый текст. Напишите следующую подпрограмму:

```
_
 Parse and Translate a Boolean Condition
 This version is a dummy
Procedure Condition;
begin
EmitLn('&lt;condition&gt;');
end;
_
```

Вставьте эту процедуру в вашу программу как раз перед DoIf. Теперь запустите программу. Испробуйте строку типа:

aibece

Как вы можете видеть, синтаксический анализатор, кажется, распознает конструкцию и вставляет объектный код в правильных местах. Теперь попробуйте набор вложенных IF:

aibicedefe

Он начинает все более походить на настоящий, не так ли?

Теперь, когда у нас есть общая идея (и инструменты такие как нотация и процедуры NewLabel и PostLabel) проще пареной репы расширить синтаксический анализатор для поддержки и других конструкций. Первое (а также и одно из самых сложных) это добавление условия ELSE в IF. В БНФ это выглядит так:

IF <condition> <block> [ELSE <block>] ENDIF

Сложность возникает просто потому, что здесь присутствует необязательное условие, которого нет в других конструкциях.

Соответствующий выходной код должен быть таким:

```
IF
&lt;condition&gt;
BEQ L1
&lt;block&gt;
BRA L2
L1: &lt;block&gt;
L2: ...
```

Это приводит нас к следующей синтаксически управляемой схеме перевода:

```
IF
&lt;condition&gt;  L1 = NewLabel;
L2 = NewLabel;
Emit(BEQ L1)
&lt;block&gt;
ELSE  Emit(BRA L2);
PostLabel(L1)
&lt;block&gt;
ENDIF  PostLabel(L2)
```

Сравнение этого со случаем IF без ELSE дает нам понимание того, как обрабатывать обе эти ситуации. Код ниже выполняет это. (Обратите внимание, что использую "l" вместо «ELSE» так как "e" имеет другое назначение):

```
–
 Recognize and Translate an IF Construct
procedure DoIf;
var L1, L2: string;
begin
Match('i');
Condition;
L1 := NewLabel;
L2 := L1;
EmitLn('BEQ ' + L1);
Block;
if Look = 'l' then begin
Match('l');
L2 := NewLabel;
EmitLn('BRA ' + L2);
PostLabel(L1);
Block;
end;
Match('e');
PostLabel(L2);
end;
–
```

Вы получили его. Законченый анализатор/транслятор в 19 строк кода.

Сейчас протестируйте его. Испробуйте что-нибудь типа:

aiblcede

Работает? Теперь, только для того, чтобы убедиться, что мы ничего не испортили и случай с IF без ELSE тоже будет обрабатываться, введите

aibece

Теперь испробуйте несколько вложенных IF. Испытайте что-нибудь на ваш выбор, включая несколько неправильных утверждений. Только запомните, что 'e' не является допустимым оператором «other».

ОПЕРАТОР WHILE

Следующий вид оператора должен быть простым, так как мы уже имеем опыт. Синтаксис, который я выбрал для оператора WHILE следующий:

WHILE <condition> <block> ENDWHILE

Знаю, знаю, мы действительно не нуждаемся в отдельных видах ограничителей для каждой конструкции... вы можете видеть, что фактически в нашей односимвольной версии 'e' используется для всех из них. Но я также помню множество сессий отладки в Паскале, пытаясь отследить своенравный END который по мнению компилятора я хотел поместить где-нибудь еще. По своему опыту знаю, что специфичные и уникальные ключевые слова, хотя они и добавляются к словарю языка, дают небольшую защиту от ошибок, которая стоит дополнительной работы создателей компиляторов.

Теперь рассмотрите, во что должен траслироваться WHILE:

L1: <condition>

BEQ L2

<block>

BRA L1

L2:

Как и прежде, сравнение этих двух представлений дает нам действия, необходимые на каждом этапе:

WHILE L1 = NewLabel;
PostLabel(L1)
<condition> Emit(BEQ L2)
<block>
ENDWHILE Emit(BRA L1);
PostLabel(L2)

Код выходит непосредственно из синтаксиса:

```
{
 Parse and Translate a WHILE Statement
procedure DoWhile;
var L1, L2: string;
begin
Match('w');
L1 := NewLabel;
L2 := NewLabel;
PostLabel(L1);
Condition;
EmitLn('BEQ ' + L2);
Block;
Match('e');
EmitLn('BRA ' + L1);
PostLabel(L2);
end;
}
```

Так как мы получили новый оператор, мы должны добавить его вызов в процедуру Block:

```
{
 Recognize and Translate a Statement Block
procedure Block;
begin
while not(Look in ['e', 'l']) do begin
case Look of
'i': DoIf;
'w': DoWhile;
else Other;
end;
end;
end;
}
```

Никаких других изменений не требуется.

Хорошо, протестируйте новую программу. Заметьте, что на этот раз код <condition> находится внутри верхней метки, как раз там, где нам надо. Попробуйте несколько вложенных циклов. Испробуйте циклы внутри IF и IF внутри циклов. Если вы немного напутаете то, что вы должны набирать, не смущайтесь: вы пишите ошибки и в других языках, не правда ли? Код будет выглядеть более осмысленным, когда мы получим полные ключевые слова.

Я надеюсь, что к настоящему времени вы начинаете понимать, что это действительно просто. Все, что нам необходимо было сделать для того, чтобы создать новую конструкцию,

это разработать ее синтаксически-управляемый перевод. Код возникает из него, и это не влияет на другие подпрограммы. Как только вы почувствуете это, вы увидите, что можете добавлять новые конструкции почти также быстро, как вы можете их придумывать.

ОПЕРАТОР LOOP

Мы могли бы остановиться на этом и иметь работающий язык. Много раз было показано, что языка высокого уровня всего с двумя конструкциями IF и WHILE достаточно для написания структурного кода. Но раз уж мы начали, то давайте немного расширим репертуар.

Эта конструкция даже проще, так как она совсем не имеет проверки условия... это бесконечный цикл. Имеет ли смысл такой цикл? Немного сам по себе, но позднее мы собираемся добавить команду BREAK, которая даст нам способ выхода из цикла. Она делает язык значительно более богатым, чем Паскаль, который не имеет команды выхода из цикла и также позволяет избежать забавных конструкций типа WHILE(1) или WHILE TRUE в C и Паскале.

Синтаксис прост:

LOOP <block> ENDLOOP

Синтаксически управляемый перевод:

LOOP L = NewLabel;
PostLabel(L)
<block>
ENDLOOP Emit(BRA L

Соответствующий код показан ниже. Так как мы уже использовали "l" для ELSE на этот раз я использовал последнюю букву "p" как «ключевое слово».

```
–
 Parse and Translate a LOOP Statement
procedure DoLoop;
var L: string;
begin
Match('p');
L := NewLabel;
PostLabel(L);
Block;
Match('e');
EmitLn('BRA ' + L);
end;
–
```

После того, как вы вставите эту подпрограмму, не забудьте добавить строчку в Block для ее вызова.

REPEAT-UNTIL

Имеется одна конструкция, которую я взял напрямую из Паскаля. Синтаксис:

REPEAT <block> UNTIL <condition>

и синтаксически-управляемый перевод:

REPEAT L = NewLabel;
PostLabel(L)
<block>
UNTIL
<condition> Emit(BEQ L)

Как обычно, код вытекает отсюда довольно легко:

```
–
```

```
 Parse and Translate a REPEAT Statement
procedure DoRepeat;
var L: string;
begin
Match('r');
L := NewLabel;
PostLabel(L);
Block;
Match('u');
Condition;
EmitLn('BEQ ' + L);
end;
–
```

Как и прежде, мы должны добавить вызов DoRepeat в Block. Хотя на этот раз есть различия. Я решил использовать "r" вместо REPEAT (естественно), но я также решил использовать "u" вместо UNTIL. Это означает, что "u" должен быть добавлен к множеству символов в условии while. Это символы, которые сигнализируют о выходе из текущего блока... символы «follow», на жаргоне разработчиков компиляторов.

```
–
 Recognize and Translate a Statement Block
procedure Block;
begin
while not(Look in ['e', 'l', 'u']) do begin
case Look of
'i': DoIf;
'w': DoWhile;
'p': DoLoop;
'r': DoRepeat;
else Other;
end;
end;
end;
–
```

ЦИКЛ FOR

Цикл FOR очень удобен, но он тяжел для трансляции. Не столько потому, что сама конструкция трудна... в конце концов это всего лишь цикл... но просто потому, что она трудна для реализации на ассемблере. Как только код придуман, трансляция достаточно проста.

Фаны Си любят цикл FOR этого языка (фактически он проще для кодирования), но вместо него я выбрал синтаксис очень похожий на синтаксис из старого доброго Бейсика:

FOR <ident> = <expr1> TO <expr2> <block> ENDFOR

Сложность трансляции цикла «FOR» зависит от выбранного вами способа его реализации, от пути, которым вы решили определять правила обработки ограничений. Рассчитывается ли expr2 каждый раз при прохождении цикла, например, или оно обрабатывается как постоянное ограничение? Всегда ли вы проходите цикл хотя бы раз, как в Fortran, или нет. Все становится проще, если вы приверженец точки зрения что эта конструкция эквивалентна:

<ident> = <expr1>
TEMP = <expr2>
WHILE <ident> <= TEMP

<block>
ENDWHILE

Заметьте, что с этим определением цикла <block> не будет выполнен вообще если <expr1> изначально больше чем <expr2>.

Код 68000, необходимый для этого, сложней чем все что мы делали до сих пор. Я сделал несколько попыток, помещая и счетчик и верхний предел в стек, в регистры и т.д. В конечном итоге я остановился на гибридном варианте размещения, при котором счетчик помещается в памяти (поэтому он может быть доступен внутри цикла) а верхний предел – в стеке. Оттранслированный код получился следующий:

```
&lt;ident&gt; ; получить имя счетчика цикла
&lt;expr1&gt; ; получить начальное значение
LEA &lt;ident&gt;(PC),A0 ; обратиться к счетчику цикла
SUBQ #1,D0 ; предварительно уменьшить его
MOVE D0,(A0) ; сохранить его
&lt;expr1&gt; ; получить верхний предел
MOVE D0,-(SP) ; сохранить его в стеке
L1: LEA &lt;ident&gt;(PC),A0 ; обратиться к счетчику цикла
MOVE (A0),D0 ; извлечь его в D0
ADDQ #1,D0 ; увеличить счетчик
MOVE D0,(A0) ; сохранить новое значение
CMP (SP),D0 ; проверить диапазон
BLE L2 ; пропустить если D0 &gt; (SP)
&lt;block&gt;
BRA L1 ; цикл для следующего прохода
L2: ADDQ #2,SP ; очистить стек
```

Ничего себе! Это же куча кода... строка, содержащая <block> кажется совсем потерявшейся. Но это лучшее из того, что я смог придумать. Я полагаю, чтобы вам помочь, вы должны иметь в виду что в действительности это всего лишь шестнадцать слов, в конце концов. Если кто-нибудь сможет оптимизировать это лучше, пожалуйста дайте мне знать.

Однако, подпрограмма анализа довольно проста теперь, когда у нас есть код:

```
{
 Parse and Translate a FOR Statement
}
procedure DoFor;
var L1, L2: string;
Name: char;
begin
Match('f');
L1 := NewLabel;
L2 := NewLabel;
Name := GetName;
Match('=');
Expression;
EmitLn('SUBQ #1,D0');
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE D0,(A0)');
Expression;
EmitLn('MOVE D0,-(SP)');
PostLabel(L1);
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE (A0),D0');
EmitLn('ADDQ #1,D0');
EmitLn('MOVE D0,(A0)');
```

```
EmitLn('CMP (SP),D0');
EmitLn('BGT ' + L2);
Block;
Match('e');
EmitLn('BRA ' + L1);
PostLabel(L2);
EmitLn('ADDQ #2,SP');
end;
–
```

Так как в этой версии синтаксического анализатора у нас нет выражений, я использовал тот же самый прием что и для Condition и написал подпрограмму:

```
–
 Parse and Translate an Expression
 This version is a dummy
Procedure Expression;
begin
EmitLn('&lt;expr&gt;');
end;
–
```

Испытайте его. Снова, не забудьте добавить вызов в Block. Так как у нас нет возможности ввода для фиктивной версии Expression, типичная входная строка будет выглядеть так:

afi=bece

Хорошо, генерируется много кода, не так ли? Но, по крайней мере, это правильный код.

ОПЕРАТОР DO

Из-за всего этого мне захотелось иметь более простую версию цикла FOR. Причина появления всего этого кода выше состоит в необходимости иметь счетчик цикла, доступный как переменная внутри цикла. Если все, что нам нужно это считающий цикл, позволяющий нам выполнить что-то определенное число раз, но не нужен непосредственный доступ к счетчику, имеется более простое решение. Процессор 68000 имеет встроенную команду «уменьшить и переход если не ноль», которая является идеальной для подсчета. Для полноты давайте добавим и эту конструкцию. Это будет наш последний цикл.

Синтаксис и его перевод:

DO
<expr> Emit(SUBQ #1,D0);
L = NewLabel;
PostLabel(L);
Emit(MOVE D0,-(SP)
<block>
ENDDO Emit(MOVE (SP)+,D0;
Emit(DBRA D0,L)

Это гораздо проще! Цикл будет выполняться <expr> раз. Вот код:

```
–
 Parse and Translate a DO Statement
procedure Dodo;
var L: string;
begin
Match('d');
```

```
L := NewLabel;
Expression;
EmitLn('SUBQ #1,D0');
PostLabel(L);
EmitLn('MOVE D0,-(SP)');
Block;
EmitLn('MOVE (SP)+,D0');
EmitLn('DBRA D0,' + L);
end;
_
```

Я думаю вы согласитесь, что это гораздо проще, чем классический цикл FOR. Однако, каждая конструкция имеет свое назначение.

ОПЕРАТОР BREAK

Ранее я обещал вам оператор BREAK для сопровождения цикла LOOP. Им я в некотором роде горд. На первый взгляд BREAK кажется действительно сложным. Моим первым подходом было просто использовать его как дополнительный ограничитель в Block и разделить все циклы на две части точно также как я сделал это для ELSE оператора IF. Но, оказывается, это не работает, потому что оператор BREAK редко находится на том же самом уровне, что и сам цикл. Наиболее вероятное место для BREAK – сразу после IF, что приводило бы к выходу из конструкции IF, а не из окружающего цикла. Неправильно. BREAK должен выходить из внутреннего LOOP даже если он вложен в несколько уровней IF.

Моей следующей мыслью было просто сохранять в какой-то глобальной переменной, метку окончания самого вложенного цикла. Это также не работает, потому что может возникнуть прерывание из внутренного цикла с последующим прерыванием из внешнего. Сохранение метки для внутреннего цикла затерло бы метку для внешного. Так глобальная переменная превратилась в стек. Дело становилось грязным.

Тогда я решил последовать своему собственному совету. Помните последний урок, когда я показал вам как хорошо служит нам неявный стек синтаксического анализатора с рекурсивным спуском. Я сказал, что если вы начинаете видеть потребность во внешнем стеке, возможно вы делаете что-то неправильно. Действительно возможно заставить рекурсию, встроенную в наш синтаксический анализатор, позаботиться обо всем и это решение настолько простое, что кажется удивительным.

Секрет состоит в том, чтобы заметить, что каждый оператор BREAK должен выполняться внутри блока... и ни в каком другом месте. Так что все, что мы должны сделать это передать в Block адрес выхода из самого внутреннего цикла. Затем он может передать этот адрес подпрограмме, транслирующей инструкцию Break. Так как оператор IF не изменяет уровень цикла, процедура DoIf не должна делать что-либо за исключением передачи метки в ее блок (оба из них). Так как циклы изменяют уровень, каждый цикл просто игнорирует любую метку выше его и передает свою собственную метку выхода дальше.

Все это проще показать вам чем описывать. Я продемонстрирую это с самым простым циклом, циклом LOOP:

```
_
 Parse and Translate a LOOP Statement
procedure DoLoop;
var L1, L2: string;
begin
Match('p');
L1 := NewLabel;
```

```
L2 := NewLabel;
PostLabel(L1);
Block(L2);
Match('e');
EmitLn('BRA ' + L1);
PostLabel(L2);
end;
–
```

Заметьте, что теперь DoLoop имеет две метки а не одну. Вторая дает команде BREAK адрес перехода Если в цикле нет BREAK, то мы зря потратили метку и немного загромоздили код, но не нанесли никакого вреда.

Заметьте также, что процедура Block теперь имеет параметр, который для циклов всегда будет адресом выхода. Новая версия Block:

```
–
 Recognize and Translate a Statement Block
procedure Block(L: string);
begin
while not(Look in ['e', 'l', 'u']) do begin
case Look of
'i': DoIf(L);
'w': DoWhile;
'p': DoLoop;
'r': DoRepeat;
'f': DoFor;
'd': DoDo;
'b': DoBreak(L);
else Other;
end;
end;
end;
–
```

Снова заметьте, что все что Block делает с меткой это передает ее в DoIf и DoBreak. Циклы не нуждаются в ней, потому что они в любом случае передают свою собственную метку.

Новая версия DoIf:

```
–
 Recognize and Translate an IF Construct
procedure Block(L: string); Forward;
procedure DoIf(L: string);
var L1, L2: string;
begin
Match('i');
Condition;
L1 := NewLabel;
L2 := L1;
EmitLn('BEQ ' + L1);
Block(L);
if Look = 'l' then begin
Match('l');
L2 := NewLabel;
EmitLn('BRA ' + L2);
PostLabel(L1);
```

```
Block(L);
end;
Match('e');
PostLabel(L2);
end;
–
```

Здесь единственное, что изменяется, это добавляется параметр у процедуры Block. Оператор IF не меняет уровень вложенности цикла, поэтому DoIf просто передает метку дальше. Независимо от того, сколько уровней вложенности IF мы имеем, будет использоваться та же самая метка.

Теперь не забудьте, что DoProgram также вызывает Block и теперь необходимо передавать ей метку. Попытка выхода из внешнего блока является ошибкой, поэтому DoProgram передает пустую метку, которая перехватывается DoBreak:

```
–
 Recognize and Translate a BREAK
procedure DoBreak(L: string);
begin
Match('b');
if L &lt;&gt; '' then
EmitLn('BRA ' + L)
else Abort('No loop to break from');
end;
–
 Parse and Translate a Program
procedure DoProgram;
begin
Block('');
if Look &lt;&gt; 'e' then Expected('End');
EmitLn('END')
end;
–
```

Этот код позаботится почти обо всем. Испытайте его, посмотрите, сможете ли вы «сломать» («break») его (каламбур). Аккуратней однако. К настоящему времени мы использовали так много букв, что трудно придумать символ, который не представляет сейчас какое либо зарезервированное слово. Не забудьте, перед тем, как вы протестируете программу, вы должны будете исправить каждый случай появления Block в других циклах для включения нового параметра. Сделайте это точно так же, как я сделал это для LOOP.

Я сказал выше «почти». Есть одна небольшая проблема: если вы внимательно посмотрите на код, генерируемый для DO, вы увидите, что если вы прервете этот цикл, то значение счетчика все еще остается в стеке. Мы должны исправить это! Позор... это была одна из самых маленьких наших подпрограмм, но это не помогло. Вот новая версия, которая не имеет этой проблемы:

```
–
 Parse and Translate a DO Statement
procedure Dodo;
var L1, L2: string;
begin
Match('d');
L1 := NewLabel;
L2 := NewLabel;
Expression;
```

```
EmitLn('SUBQ #1,D0');
PostLabel(L1);
EmitLn('MOVE D0,-(SP)');
Block(L2);
EmitLn('MOVE (SP)+,D0');
EmitLn('DBRA D0,' + L1);
EmitLn('SUBQ #2,SP');
PostLabel(L2);
EmitLn('ADDQ #2,SP');
end;
{--------------------------------------------------------------}
```

Две дополнительные инструкции SUBQ и ADDQ заботятся о сохранении стека в правильной форме.

ЗАКЛЮЧЕНИЕ

К этому моменту мы создали ряд управляющих конструкций... в действительности более богатый набор чем предоставляет почти любой другой язык программирования. И, за исключением цикла FOR, это было довольно легко сделать. Но даже этот цикл был сложен только потому, что сложность заключалась в ассемблере.

Я завершаю на этом урок. Чтобы можно было обернуть наш продукт красной ленточкой, в действительности мы должны иметь настоящие ключевые слова вместо этих игрушечных односимвольных. Вы уже видели, что расширить компилятор для поддержки многосимвольных слов не трудно, но в этом случае возникнут большие различия в представлении нашего входного кода. Я оставлю этот небольшой кусочек для следующей главы. В этой главе мы также рассмотрим логические выражения, так что мы сможем избавиться от фиктивной версии Condition, которую мы здесь использовали. Увидимся.

Для справочных целей привожу полный текст синтаксического анализатора для этого урока:

```
{--------------------------------------------------------------}
program Branch;
{--------------------------------------------------------------}
{ Constant Declarations }
const TAB = ^I;
CR = ^M;
{--------------------------------------------------------------}
{ Variable Declarations }
var Look : char;  { Lookahead Character }
Lcount: integer;  { Label Counter }
{--------------------------------------------------------------}
{ Read New Character From Input Stream }
procedure GetChar;
begin
Read(Look);
end;
{--------------------------------------------------------------}
{ Report an Error }
procedure Error(s: string);
begin
WriteLn;
WriteLn(^G, 'Error: ', s, '.');
end;
{--------------------------------------------------------------}
```

```
{ Report Error and Halt }
procedure Abort(s: string);
begin
Error(s);
Halt;
end;
{--}
{ Report What Was Expected }
procedure Expected(s: string);
begin
Abort(s + ' Expected');
end;
{--}
{ Match a Specific Input Character }
procedure Match(x: char);
begin
if Look = x then GetChar
else Expected('"' + x + '"');
end;
{--}
{ Recognize an Alpha Character }
function IsAlpha(c: char): boolean;
begin
IsAlpha := UpCase(c) in ['A'..'Z'];
end;
{--}
{ Recognize a Decimal Digit }
function IsDigit(c: char): boolean;
begin
IsDigit := c in ['0'..'9'];
end;
{--}
{ Recognize an Addop }
function IsAddop(c: char): boolean;
begin
IsAddop := c in ['+', '-'];
end;
{--}
{ Recognize White Space }
function IsWhite(c: char): boolean;
begin
IsWhite := c in [' ', TAB];
end;
{--}
{ Skip Over Leading White Space }
procedure SkipWhite;
begin
while IsWhite(Look) do
GetChar;
end;
{--}
{ Get an Identifier }
function GetName: char;
begin
if not IsAlpha(Look) then Expected('Name');
GetName := UpCase(Look);
GetChar;
```

```
end;
_
 Get a Number
function GetNum: char;
begin
if not IsDigit(Look) then Expected('Integer');
GetNum := Look;
GetChar;
end;
_
 Generate a Unique Label
function NewLabel: string;
var S: string;
begin
Str(LCount, S);
NewLabel := 'L' + S;
Inc(LCount);
end;
_
 Post a Label To Output
procedure PostLabel(L: string);
begin
WriteLn(L, ':');
end;
_
 Output a String with Tab
procedure Emit(s: string);
begin
Write(TAB, s);
end;
_
 Output a String with Tab and CRLF
procedure EmitLn(s: string);
begin
Emit(s);
WriteLn;
end;
_
 Parse and Translate a Boolean Condition
procedure Condition;
begin
EmitLn('&lt;condition&gt;');
end;
_
 Parse and Translate a Math Expression
procedure Expression;
begin
EmitLn('&lt;expr&gt;');
end;
_
 Recognize and Translate an IF Construct
procedure Block(L: string); Forward;
procedure DoIf(L: string);
var L1, L2: string;
begin
Match('i');
Condition;
```

```
L1 := NewLabel;
L2 := L1;
EmitLn('BEQ ' + L1);
Block(L);
if Look = 'l' then begin
Match('l');
L2 := NewLabel;
EmitLn('BRA ' + L2);
PostLabel(L1);
Block(L);
end;
Match('e');
PostLabel(L2);
end;
{--------------------------------------------------------------}
{ Parse and Translate a WHILE Statement }
procedure DoWhile;
var L1, L2: string;
begin
Match('w');
L1 := NewLabel;
L2 := NewLabel;
PostLabel(L1);
Condition;
EmitLn('BEQ ' + L2);
Block(L2);
Match('e');
EmitLn('BRA ' + L1);
PostLabel(L2);
end;
{--------------------------------------------------------------}
{ Parse and Translate a LOOP Statement }
procedure DoLoop;
var L1, L2: string;
begin
Match('p');
L1 := NewLabel;
L2 := NewLabel;
PostLabel(L1);
Block(L2);
Match('e');
EmitLn('BRA ' + L1);
PostLabel(L2);
end;
{--------------------------------------------------------------}
{ Parse and Translate a REPEAT Statement }
procedure DoRepeat;
var L1, L2: string;
begin
Match('r');
L1 := NewLabel;
L2 := NewLabel;
PostLabel(L1);
Block(L2);
Match('u');
Condition;
EmitLn('BEQ ' + L1);
```

```
PostLabel(L2);
end;
_
 Parse and Translate a FOR Statement
procedure DoFor;
var L1, L2: string;
Name: char;
begin
Match('f');
L1 := NewLabel;
L2 := NewLabel;
Name := GetName;
Match('=');
Expression;
EmitLn('SUBQ #1,D0');
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE D0,(A0)');
Expression;
EmitLn('MOVE D0,-(SP)');
PostLabel(L1);
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE (A0),D0');
EmitLn('ADDQ #1,D0');
EmitLn('MOVE D0,(A0)');
EmitLn('CMP (SP),D0');
EmitLn('BGT ' + L2);
Block(L2);
Match('e');
EmitLn('BRA ' + L1);
PostLabel(L2);
EmitLn('ADDQ #2,SP');
end;
_
 Parse and Translate a DO Statement
procedure Dodo;
var L1, L2: string;
begin
Match('d');
L1 := NewLabel;
L2 := NewLabel;
Expression;
EmitLn('SUBQ #1,D0');
PostLabel(L1);
EmitLn('MOVE D0,-(SP)');
Block(L2);
EmitLn('MOVE (SP)+,D0');
EmitLn('DBRA D0,' + L1);
EmitLn('SUBQ #2,SP');
PostLabel(L2);
EmitLn('ADDQ #2,SP');
end;
_
 Recognize and Translate a BREAK
procedure DoBreak(L: string);
begin
Match('b');
EmitLn('BRA ' + L);
```

```
end;
–
 Recognize and Translate an «Other»
procedure Other;
begin
EmitLn(GetName);
end;
–
 Recognize and Translate a Statement Block
procedure Block(L: string);
begin
while not(Look in ['e', 'l', 'u']) do begin
case Look of
'i': DoIf(L);
'w': DoWhile;
'p': DoLoop;
'r': DoRepeat;
'f': DoFor;
'd': DoDo;
'b': DoBreak(L);
else Other;
end;
end;
end;
–
 Parse and Translate a Program
procedure DoProgram;
begin
Block('');
if Look &lt;&gt; 'e' then Expected('End');
EmitLn('END')
end;
–
 Initialize
procedure Init;
begin
LCount := 0;
GetChar;
end;
–
 Main Program
begin
Init;
DoProgram;
end.
–
```

Булевы выражения

ВВЕДЕНИЕ

В пятой части этой серии мы рассмотрели управляющие конструкции и разработали подпрограммы синтаксического анализа для трансляции их в объектный код. Мы закончили с хорошим, относительно богатым набором конструкций.

Однако, когда мы оставили синтаксический анализатор, в наших возможностях существовал один большой пробел: мы не обращались к вопросу условия ветвления. Чтобы заполнить пустоту, я представил вам фиктивную подпрограмму анализа Condition, которая служила только как заменитель настоящей.

Одним из дел, которыми мы займемся на этом уроке, будет заполнение этого пробела посредством расширения Condition до настоящего анализатора/транслятора.

ПЛАН

Мы собираемся подойти к этой главе немного по-другому, чем к любой другой. В других главах мы начинали немедленно с экспериментов, используя компилятор Pascal, выстраивая синтаксические анализаторы от самых элементарных начал до их конечных форм, не тратя слишком много времени на предварительное планирование. Это называется кодированием без спецификации и обычно к нему относятся неодобрительно. Раньше мы могли избегать планирования, потому что правила арифметики довольно хорошо установлены... мы знаем, что означает знак "+" без необходимости подробно это обсуждать. То же самое относится к ветвлениям и циклам. Но способы, которыми языки программирования реализуют логику, немного отличаются от языка к языку. Поэтому прежде, чем мы начнем серьезное кодирование, лучше мы сперва примем решение что же мы хотим. И способ сделать это находится на уровне синтаксических правил БНФ (грамматики).

ГРАММАТИКА

Некоторое время назад мы реализовали синтаксические уравнения БНФ для арифметических выражений фактически даже не записав их все в одном месте. Пришло время сделать это. Вот они:

<expression> ::= <unary op> <term> [<addop> <term>]*

<term> ::= <factor> [<mulop> factor]*

<factor> ::= <integer> | <variable> | (<expression>)

(Запомните, преимущества этой грамматики в том, что она осуществляет такую иерархию приоритетов операторов, которую мы обычно ожидаем для алгебры.)

На самом деле, пока мы говорим об этом, я хотел бы прямо сейчас немного исправить эту грамматику. Способ, которым мы обрабатываем унарный минус, немного неудобный. Я нашел, что лучше записать грамматику таким образом:

<expression> ::= <term> [<addop> <term>]*

<term> ::= <signed factor> [<mulop> factor]*

<signed factor> ::= [<addop>] <factor>

<factor> ::= <integer> | <variable> | (<expression>)

Это возлагает обработку унарного минуса на Factor, которому он в действительности и принадлежит.

Это не означает, что вы должны возвратиться назад и переписать программы, которые вы уже написали, хотя вы свободны сделать так, если хотите. Но с этого момента я буду использовать новый синтаксис.

Теперь, возможно, для вас не будет ударом узнать, что мы можем определить аналогичную грамматику для булевой алгебры. Типичный набор правил такой:

<b-expression>::= <b-term> [<orop> <b-term>]*

<b-term> ::= <not-factor> [AND <not-factor>]*

<not-factor> ::= [NOT] <b-factor>

<b-factor> ::= <b-literal> | <b-variable> | (<b-expression>)

Заметьте, что в этой грамматике оператор AND аналогичен "*", а OR (и исключающее OR) – "+". Оператор NOT аналогичен унарному минусу. Эта иерархия не является

абсолютным стандартом... некоторые языки, особенно Ada, обрабатывают все логические операторы как имеющие одинаковый уровень приоритета... но это кажется естественным.

Обратите также внимание на небольшое различие способов, которыми обрабатываются NOT и унарный минус. В алгебре унарный минус считается идущим со всем термом и поэтому никогда не появляется более одного раза в данном терме. Поэтому выражение вида:

a * -b

или еще хуже:

a – -b

не разрешены. В булевой алгебре наоборот, выражение

a AND NOT b

имеет точный смысл и показанный синтаксис учитывает это.

ОПЕРАТОРЫ ОТНОШЕНИЙ

Итак, предполагая что вы захотите принять грамматику, которую я показал здесь, мы теперь имеем синтаксические правила и для арифметики и для булевой алгебры. Сложность возникает когда мы должны объединить их. Почему мы должны сделать это? Ну, эта тема возникла из-за необходимости обрабатывать «предикаты» (условия), связанные с управляющими операторами такими как IF. Предикат должен иметь логическое значение, то есть он должен быть оценен как TRUE или FALSE. Затем ветвление выполняется или не выполняется в зависимости от этого значения. Тогда то, что мы ожидаем увидеть происходящим в процедуре Condition, будет вычисление булевого выражения.

Но имеется кое-что еще. Настоящее булево выражение может действительно быть предикатом управляющего оператора... подобно:

IF a AND NOT b THEN ....

Но более часто мы видим, что булева алгебра появляется в таком виде:

IF (x >= 0) and (x <= 100) THEN...

Здесь два условия в скобках являются булевыми выражениями, но индивидуальные сравниваемые термы: x, 0 и 100 являются числовыми по своей природе. Операторы отношений >= и <= являются катализаторами, с помощью которых булевские и арифметические компоненты объединяются вместе.

Теперь, в примере выше сравниваемые термы являются просто термами. Однако, в общем случае, каждая сторона может быть математическим выражением. Поэтому мы можем определить отношение как:

<relation> ::= <expression> <relop> <expression>,

где выражения, о которых мы говорим здесь – старого числового типа, а операторы отношений это любой из обычных символов:

=, <> (или !=), <, >, <= и >=

Если вы подумаете об этом немного, то согласитесь, что так как этот вид предиката имеет логическое значение, TRUE или FALSE, это в действительности просто еще один вид показателя. Поэтому мы можем расширить определение булевого показателя следующим образом:

<b-factor> ::= <b-literal>
| <b-variable>
| (<b-expression>)
| <relation>

Вот эта связь! Операторы отношений и отношения, которые они определяют, служат для объединения двух типов алгебры. Нужно заметить, что это подразумевает иерархию, в которой арифметическое выражение имеет более высокий приоритет, чем булевский показатель и, следовательно, чем все булевы операторы. Если вы выпишите уровни приоритета для всех операторов, вы прийдете к следующему списку:

Уровень Синтаксический элемент Оператор

0 factor literal, variable
1 signed factor unary minus
2 term *, /
3 expression +, -
4 b-factor literal, variable, relop
5 not-factor NOT
6 b-term AND
7 b-expression OR, XOR

Если мы захотим принять столько уровней приоритета, эта грамматика кажется приемлемой. К несчастью, она не будет работать! Грамматика может быть великолепной в теории, но она может совсем не иметь смысла в практике нисходящего синтаксического анализатора. Чтобы увидеть проблему рассмотрите следующий фрагмент кода:

IF ((((((A + B + C) < 0) AND....

Когда синтаксический анализатор анализирует этот код он знает, что после того, как он рассмотрит токен IF следующим должно быть булево выражение. Поэтому он может приступить к началу вычисления такого выражения. Но первое выражение в примере является арифметическим выражением A + B + C. Хуже того, в точке, в которой анализатор прочитал значительную часть входной строки:

IF ((((((A ,

он все еще не имеет способа узнать с каким видом выражения он имеет дело. Так не пойдет, потому что мы должны иметь две различных программы распознавания для этих двух случаев. Ситуация может быть обработана без изменения наших определений но только если мы захотим принять произвольное количество возвратов (backtracking) чтобы избавить наш путь от неверных предположений. Ни один из создателей компиляторов в здравом уме не согласился бы на это.

Происходит то, что красота и элегантность грамматики БНФ столкнулась лицом к лицу с реальностью технологии компиляции.

Чтобы работать с этой ситуацией создатели компиляторов должны идти на компромиссы, так чтобы один анализатор мог бы поддерживать грамматику без возвратов.

ИСПРАВЛЕНИЕ ГРАММАТИКИ

Проблема, с которой мы столкнулись, возникает потому, что наше определение и арифметических и булевых показателей позволяет использовать выражения в скобках. Так как определения рекурсивны, мы можем закончить с любым числом уровней скобок и синтаксический анализатор не может знать с каким видом выражения он имеет дело.

Решение просто, хотя и приводит к глубоким изменениям нашей грамматики. Мы можем разрешить круглые скобки только в одном виде показателей. Способ сделать это значительно изменяется от языка к языку. Это то место, где не существует соглашения или договора способного нам помочь.

Когда Никлаус Вирт разработал Паскаль, его желанием было ограничить количество уровней приоритета (меньше подпрограмм синтаксического анализа, в конце концов). Так операторы OR и исключающее OR рассматриваются просто как Addop и обрабатываются на уровне математического выражения. Аналогично AND рассматривается подобно Mulop и обрабатывается с Term. Уровни приоритета:

Заметьте, что имеется только один набор синтаксических правил, применимый к обоим видам операторов. Тогда согласно этой грамматике выражения типа:

x + (y AND NOT z) DIV 3

являются совершенно допустимыми. И, фактически, они таковыми являются... настолько, насколько синтаксический анализатор в этом заинтересован. Паскаль не

позволяет смешивать арифметические и логические переменные, и подобные вещи скорее перехватываются на семантическом уровне, когда придет время генерировать для них код, чем на синтаксическом уровне.

Авторы C взяли диаметрально противоположный метод: они обрабатывают операторы как разные и C имеет что-то гораздо более похожее на наши семь уровней приоритета. Фактически, в C имеется не менее 17 уровней! Дело в том, что C имеет также операторы '=', '+=' и их родственников '<<', '>>', '++', '–' и т.д. Как ни странно, хотя в C арифметические и булевые операторы обрабатываются раздельно, то переменные нет... в C нет никаких булевых или логических переменных, так что логическая проверка может быть сделана на любом целочисленном значении.

Мы сделаем нечто среднее. Я склонен обычно придерживаться Паскалевского подхода, так как он кажется самым простым с точки зрения реализации, но это приводит к некоторым странностям, которые я никогда очень сильно не любил, как например в выражении:

IF (c >= 'A') and (c <= 'Z') then ...

скобки обязательны. Я никогда не мог понять раньше почему, и ни мой компилятор, ни любой человек также не объясняли этого достаточно хорошо. Но сейчас мы все можем видеть, что оператор «and», имеющий приоритет как у оператора умножения, имеет более высокий приоритет, чем у операторов отношения, поэтому без скобок выражение эквивалентно:

IF c >= ('A' and c) <= 'Z' then

что не имеет смысла.

В любом случае, я решил разделить операторы на различные уровни, хотя и не столько много как в C.

<b-expression> ::= <b-term> [<orop> <b-term>]*
<b-term> ::= <not-factor> [AND <not-factor>]*
<not-factor> ::= [NOT] <b-factor>
<b-factor> ::= <b-literal> | <b-variable> | <relation>
<relation> ::= | <expression> [<relop> <expression]
<expression> ::= <term> [<addop> <term>]*
<term> ::= <signed factor> [<mulop> factor]*
<signed factor>::= [<addop>] <factor>
<factor> ::= <integer> | <variable> | (<b-expression>)

Эта грамматика приводит к тому же самому набору семи уровней, которые я показал ранее. Действительно, это почти таже самая грамматика... я просто исключил заключенное в скобки b-выражение как возможный b-показатель и добавил отношение как допустимую форму b-показателя.

Есть одно тонкое, но определяющее различие, которое заставляет все это работать. Обратите внимание на квадратные скобки в определении отношения. Это означает, что relop и второе выражение являются необязательными.

Странным последствием этой грамматики (которое присутствует и в C) является то, что каждое выражения потенциально является булевым выражение. Синтаксический анализатор всегда будет искать булевское выражение, но «уладит» все до арифметического. Честно говоря, это будет замедлять синтаксический анализатор потому что он должен пройти через большее количество вызовов процедур. Это одна из причин, почему компиляторы Паскаля обычно быстрее выполяют компиляцию, чем компиляторы C. Если скорость для вас – больное место, придерживайтесь синтаксиса Паскаля.

СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР

Теперь, когда мы прошли через процесс принятия решений, мы можем поспешить с разработкой синтаксического анализатора. Вы делали это со мной несколько раз до этого, поэтому вы знаете последовательность: мы начнем с новой копии Cradle и будем добавлять

процедуры одна за другой. Так что давайте сделаем это.

Мы начинаем, как и в случае с арифметикой, работая с булевыми литералами а не с переменными. Это дает нам новый вид входного токена, поэтому нам также нужна новая программа распознавания и новая процедура для чтения экземпляров этого типа токенов. Давайте начнем, определив эти две новые процедуры:

```
{--------------------------------------------------------------}
{ Recognize a Boolean Literal }
function IsBoolean(c: char): Boolean;
begin
IsBoolean := UpCase(c) in ['T', 'F'];
end;
{--------------------------------------------------------------}
{ Get a Boolean Literal }
function GetBoolean: Boolean;
var c: char;
begin
if not IsBoolean(Look) then Expected('Boolean Literal');
GetBoolean := UpCase(Look) = 'T';
GetChar;
end;
{--------------------------------------------------------------}
```

Внесите эти подпрограммы в вашу программу. Вы можете протестировать их, добавив в основную программу оператор печати:

```
WriteLn(GetBoolean);
```

Откомпилируйте программу и протестируйте ее. Как обычно пока не очень впечатляет но скоро будет.

Теперь, когда мы работали с числовыми данными, мы должны были организовать генерацию кода для загрузки значений в D0. Нам необходимо сделать то же самое и для булевых данных. Обычным способом кодирования булевых переменных является использование 0 для представления FALSE и какого-либо другого значения для TRUE. Многие языки, как например C, используют для его представления целое число 1. Но я предпочитаю FFFF (или -1) потому что побитовое NOT также возвратит логическое NOT. Итак, нам теперь нужно выдать правильный ассемблерный код для загрузки этих значений. Первая засечка на синтаксическом анализаторе булевых выражений (BoolExpression, конечно):

```
{--------------------------------------------------------------}
{ Parse and Translate a Boolean Expression }
procedure BoolExpression;
begin
if not IsBoolean(Look) then Expected('Boolean Literal');
if GetBoolean then
EmitLn('MOVE #-1,D0')
else
EmitLn('CLR D0');
end;
{--------------------------------------------------------------}
```

Добавьте эту процедуру в ваш анализатор и вызовите ее из основной программы (заменив оператор печати, который вы только что там поместили). Как вы можете видеть, мы все еще не имеем значительной части синтаксического анализатора, но выходной код начинает выглядеть более реалистичным.

Затем, конечно, мы должны расширить определение булевого выражения. У нас уже

есть правило в БНФ:

<b-expression> ::= <b-term> [<orop> <b-term>]*

Я предпочитаю Паскалевскую версию «orop» – OR и XOR. Но так как мы сохраняем односимвольные токены, я буду кодировать их знаками '|' и '~'. Следующая версия BoolExpression – почти полная копия арифметической процедуры Expression:

```
{
 Recognize and Translate a Boolean OR
procedure BoolOr;
begin
Match('|');
BoolTerm;
EmitLn('OR (SP)+,D0');
end;
{
 Recognize and Translate an Exclusive Or
procedure BoolXor;
begin
Match('~');
BoolTerm;
EmitLn('EOR (SP)+,D0');
end;
{
 Parse and Translate a Boolean Expression
procedure BoolExpression;
begin
BoolTerm;
while IsOrOp(Look) do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'|': BoolOr;
'~': BoolXor;
end;
end;
end;
{
```

Обратите внимание на новую процедуру IsOrOp, которая также является копией, на этот раз IsAddOp:

```
{
 Recognize a Boolean Orop
function IsOrop(c: char): Boolean;
begin
IsOrop := c in ['|', '~'];
end;
{
```

ОК, переименуйте старую версию BoolExpression в BoolTerm, затем наберите код, представленный выше. Откомпилируйте и протестируйте эту версию. К этому моменту выходной код начинает выглядеть довольно хорошим. Конечно, нет большого смысла от булевой алгебры над постоянными значениями, но скоро мы расширим булевы типы, с которыми мы работаем.

Возможно вы уже предположили, какой будет следующий шаг: булевская версия Term.

Переименуйте текущую процедуру BoolTerm в NotFactor, и введите следующую новую версию BoolTerm. Заметьте, что она намного более простая, чем числовая версия, так как здесь нет эквивалента деления.

```
_
 Parse and Translate a Boolean Term
procedure BoolTerm;
begin
NotFactor;
while Look = '&amp;amp;' do begin
EmitLn('MOVE D0,-(SP)');
Match('&amp;amp;');
NotFactor;
EmitLn('AND (SP)+,D0');
end;
end;
_
```

Теперь мы почти дома. Мы транслируем сложные булевые выражения, хотя только и для постоянных значений. Следующий шаг – учесть NOT. Напишите следующую процедуру:

```
_
 Parse and Translate a Boolean Factor with NOT
procedure NotFactor;
begin
if Look = '!' then begin
Match('!');
BoolFactor;
EmitLn('EOR #-1,D0');
end
else
BoolFactor;
end;
_
```

И переименуйте предыдущую процедуру в BoolFactor. Теперь испытайте компилятор. К этому времени синтаксический анализатор должен обрабатывать любое булевое выражение, которое вы позаботитесь ему подкинуть. Работает? Отлавливает ли он неправильно сформированные выражения?

Если вы следили за тем, что мы делали в синтаксическом анализаторе для математических выражений вы знаете что далее мы расширили определение показателя для включения переменных и круглых скобок. Мы не должны делать это для булевого показателя, потому что об этих маленьких вещах позаботится наш следующий шаг. Необходима только одна дополнительная строка в BoolFactor, чтобы позаботиться об отношениях:

```
_
 Parse and Translate a Boolean Factor
procedure BoolFactor;
begin
if IsBoolean(Look) then
if GetBoolean then
EmitLn('MOVE #-1,D0')
else
EmitLn('CLR D0')
else Relation;
end;
_
```

Вы могли бы задаться вопросом, когда я собираюсь предоставить булевские переменные и булевские выражения в скобках. Отвечаю: никогда. Помните, ранее мы убрали их из грамматики. Прямо сейчас я собираюсь кодировать грамматику, которую мы уже согласовали. Сам компилятор не может видеть разницы между булевыми переменными или выражениями и арифметическими переменными или выражениями... все это будет обрабатываться в Relation в любом случае.

Конечно, понадобится некоторый код для Relation. Однако, я не чувствую себя комфортно, добавляя еще код, не проверив сперва тот, который мы уже имеем. Так что давайте сейчас просто напишем фиктивную версию Relation, которая ничего не делает за исключением того, что съедает текущий символ и выводит небольшое сообщение:

```
_
 Parse and Translate a Relation
procedure Relation;
begin
WriteLn('&lt;Relation&gt;');
GetChar;
end;
_
```

ОК, наберите этот код и испытайте его. Все старые дела все еще должны работать... у вас должна быть возможность генерировать код для AND, OR и NOT. Кроме того, если вы наберете любой алфавитный символ, вы должны получить небольшой заменитель <Relation>, где должен быть булев показатель. Вы получили это? Отлично, тогда давайте перейдем к полной версии Relation.

Чтобы получить ее, тем не менее, сначала мы должны положить небольшое основание. Вспомните, что отношение имеет форму:

<relation> ::= | <expression> [<relop> <expression]

Так как у нас появился новый вид операторов, нам также понадобится новая логическая функция для ее распознавания. Эта функция показана ниже. Из-за ограничения в один символ, я придерживаюсь четырех операторов, которые могут быть закодированы такими символами («не равно» закодировано как "#").

```
_
 Recognize a Relop
function IsRelop(c: char): Boolean;
begin
IsRelop := c in ['=', '#', '&lt;', '&gt;'];
end;
_
```

Теперь вспомните, что мы используем нуль или -1 в регистре D0 для представления логического значения и также то, что операторы цикла ожидают, что будет установлен соответствующий флаг. При реализации всего этого для 68000 все становится немного сложным.

Так как операторы цикла выполняются только по флажкам, было бы хорошо (а также довольно эффективно) просто установить эти флажки и совсем ничего не загружать в D0. Это было бы прекрасно для циклов и ветвлений, но запомните, что отношения могут быть использованы везде, где могут быть использованы булевы показатели. Мы можем сохранять его результат в булевой переменной. Так как мы не можем знаеть пока как будет использоваться результат, мы должны учесть оба случая.

Сравнение числовых данных достаточно просто... 68000 имеет команду для этого... но она устанавливает флажки а не значение. Более того, всегда будут устанавливаться те же самые флажки (ноль если равно, и т.д.), в то время, как нам необходим по-разному

установленный флажок нуля для каждого различного оператора отношения.

Решение заключается в инструкции Scc процессора 68000, которая устанавливает значение байта в 0000 или FFFF (забавно как это работает!) в зависимости от результата определенного условия. Если мы сделаем байтом результата регистр D0, мы получим необходимое логическое значение.

К сожалению, имеется одно заключительное осложнение: в отличие от почти всех других команд в наборе 68000, Scc не сбрасывает флажки условий в соответствии с сохраняемыми данными. Поэтому мы должны сделать последний шаг, проверить D0 и установить соответствующим образом флажки. Это должно быть похоже на оборот вокруг луны для получения того, что мы хотим: мы сначала выполняем проверку, затем проверяем флажки, чтобы установить данные в D0, затем тестируем D0 чтобы установить флажки снова. Это окольный путь, но это самый простой способ получить правильные флажки и, в конце концов, это всего лишь пара инструкций.

Я мог бы упомянуть здесь, что эта область, по моему мнению, показывает самые большие различия между эффективностью вручную написанного на ассемблере и сгенерированного компилятором кода. Мы уже видели, что мы теряем эффективность при арифметических операциях, хотя позже я планирую показать вам как ее немного улучшить. Мы также видели, что управляющие конструкции сами по себе могут быть реализованы довольно эффективно... обычно очень сложно улучшить код, сгенерированный для IF или WHILE. Но практически каждый компилятор, который я когда-либо видел, генерирует ужасный код, по сравнению с ассемблером, для вычисления булевых функций и особенно отношений. Причина как раз в том, о чем я упомянул выше. Когда я пишу код на ассемблере, я двигаюсь вперед и выполняю проверку наиболее удобным для меня способом, и затем подготавливаю ветвление так чтобы переход был выполнен на нужную ветку. Фактически, я «подстраиваю» каждое ветвление под ситуацию. Компилятор не может сделать этого (практически) и он также не может знать, что нам не нужно сохранять результат проверки как булевскую переменную. Поэтому он должен генерировать код по очень строгим правилам и часто заканчивает сохранением результата как булевой переменной, которая никогда не будет использована для чего-либо.

В любом случае мы теперь готовы рассмотреть код для Relation. Он показан ниже с сопровождающими процедурами:

```
_
 Recognize and Translate a Relational «Equals»
procedure Equals;
begin
Match('=');
Expression;
EmitLn('CMP (SP)+,D0');
EmitLn('SEQ D0');
end;
_
 Recognize and Translate a Relational «Not Equals»
procedure NotEquals;
begin
Match('#');
Expression;
EmitLn('CMP (SP)+,D0');
EmitLn('SNE D0');
end;
_
 Recognize and Translate a Relational «Less Than»
procedure Less;
begin
```

```
Match('&lt;');
Expression;
EmitLn('CMP (SP)+,D0');
EmitLn('SGE D0');
end;
_
 Recognize and Translate a Relational «Greater Than»
procedure Greater;
begin
Match('&gt;');
Expression;
EmitLn('CMP (SP)+,D0');
EmitLn('SLE D0');
end;
_
 Parse and Translate a Relation
procedure Relation;
begin
Expression;
if IsRelop(Look) then begin
EmitLn('MOVE D0,-(SP)');
case Look of
'=': Equals;
'#': NotEquals;
'&lt;': Less;
'&gt;': Greater;
end;
EmitLn('TST D0');
end;
end;
_
```

Теперь этот вызов Expression выглядит знакомым! Вот где редактор вашей системы оказывается полезным. Мы уже генерировали код для Expression и его близнецов на предыдущих уроках. Теперь вы можете скопировать их в ваш файл. Не забудьте использовать односимвольную версию. Просто чтобы быть уверенным, я продублировал арифметические процедуры ниже. Если вы наблюдательны, вы также увидите, что я их немного изменил чтобы привести в соответствие с последней версией синтаксиса. Эти изменения не являются необходимыми, так что вы можете предпочесть оставить все как есть до тех пор, пока не будете уверены, что все работает.

```
_
 Parse and Translate an Identifier
procedure Ident;
var Name: char;
begin
Name:= GetName;
if Look = '(' then begin
Match('(');
Match(')');
EmitLn('BSR ' + Name);
end
else
EmitLn('MOVE ' + Name + '(PC),D0');
end;
_
```

```
 Parse and Translate a Math Factor
procedure Expression; Forward;
procedure Factor;
begin
if Look = '(' then begin
Match('(');
Expression;
Match(')');
end
else if IsAlpha(Look) then
Ident
else
EmitLn('MOVE #' + GetNum + ',D0');
end;
_
 Parse and Translate the First Math Factor
procedure SignedFactor;
begin
if Look = '+' then
GetChar;
if Look = '-' then begin
GetChar;
if IsDigit(Look) then
EmitLn('MOVE #-' + GetNum + ',D0')
else begin
Factor;
EmitLn('NEG D0');
end;
end
else Factor;
end;
_
 Recognize and Translate a Multiply
procedure Multiply;
begin
Match('*');
Factor;
EmitLn('MULS (SP)+,D0');
end;
_
 Recognize and Translate a Divide
procedure Divide;
begin
Match('/');
Factor;
EmitLn('MOVE (SP)+,D1');
EmitLn('EXS.L D0');
EmitLn('DIVS D1,D0');
end;
_
 Parse and Translate a Math Term
procedure Term;
begin
SignedFactor;
while Look in ['*', '/'] do begin
EmitLn('MOVE D0,-(SP)');
case Look of
```

```
'*': Multiply;
'/': Divide;
end;
end;
end;
_
 Recognize and Translate an Add
procedure Add;
begin
Match('+');
Term;
EmitLn('ADD (SP)+,D0');
end;
_
 Recognize and Translate a Subtract
procedure Subtract;
begin
Match('-');
Term;
EmitLn('SUB (SP)+,D0');
EmitLn('NEG D0');
end;
_
 Parse and Translate an Expression
procedure Expression;
begin
Term;
while IsAddop(Look) do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'+': Add;
'-': Subtract;
end;
end;
end;
_
```

Теперь вы получили что-то... синтаксический анализатор, который может обрабатывать и арифметику и булеву алгебру и их комбинации через использование операторов отношений. Я советую вам сохранить копию этого синтаксического анализатора в безопасном месте для будущих обращений, потому что на нашем следующем шаге мы собираемся разделить его.

ОБЪЕДИНЕНИЕ С УПРАВЛЯЮЩИМИ КОНСТРУКЦИЯМИ

Сейчас давайте возвратимся назад к файлу который мы создали ранее и который выполняет синтаксический анализ управляющих конструкций. Помните небольшие фиктивные процедуры Condition и Expression? Теперь вы знаете, что в них должно находиться!

Я предупреждаю вас, вы собираетесь сделать некоторые творческие изменения, поэтому потратьте ваше время и сделайте это правильно. Вы должны скопировать все процедуры из анализатора логики от Ident до BoolExpression в синтаксический анализатор управляющих конструкций. Вставьте их в текущей позиции Condition. Затем удалите эту процедуру, так же как и фиктивную Expression. Затем замените каждый вызов Condition на обращение к BoolExpression. Наконец скопируйте процедуры IsMulop, IsOrOp, IsRelop,

IsBoolean, и GetBoolean на место. Этого достаточно.

Откомпилируйте полученную программу и протестируйте ее. Так как мы не использовали эту программу некоторое время, не забудьте, что мы использовали односимвольные токены для IF, WHILE и т.д. Также не забудьте, что любая буква, не являющаяся ключевым словом, просто отображается на экране как блок.

Попробуйте:

ia=bxlye

что означает «IF a=b X ELSE Y ENDIF».

Что вы думаете? Работает? Попробуйте что-нибудь еще.

ДОБАВЛЕНИЕ ПРИСВАИВАНИЙ

Раз у нас уже есть подпрограммы для выражений, мы могли бы также заменить «блоки» настоящими операциями присваивания. Мы уже делали это прежде, поэтому это не будет слишком трудно. Прежде, чем сделать этот шаг, однако, мы должны исправить кое-что еще.

Скоро мы обнаружим, что наши однострочные «программы», которые мы здесь пишем, будут ограничивать наш стиль. В настоящее время у нас нет способа вылечить это, потому что наш компилятор не распознает символы конца строки, возврат каретки (CR) и перевод строки (LF). Поэтому перед продвижением дальше давайте заткнем эту дыру.

Существует пара способов для работы с CR/LF. Один (подход C/Unix) просто рассматривает их как дополнительные символы пробела и игнорирует их. Фактически это не такой плохой подход, но он приводит к странным результатам для нашего анализатора в его текущем состоянии. Если бы он считывал входной поток из исходного файла как любой уважающий себя настоящий компилятор, не было бы никаких проблем. Но мы считываем входной поток с клавиатуры и ожидаем, что должно что-то произойти, когда мы нажимаем клавишу Return. Этого не произойдет, если мы просто перескакиваем CR и LF (попробуйте это). Поэтому я собираюсь использовать здесь другой метод, который в конечном счете не обязательно является лучшим методом. Рассматривайте его как временную замену до тех пор, пока мы не двинемся дальше.

Вместо того, чтобы пропускать CR/LF, мы позволим синтаксическому анализатору двигаться вперед и отлавливать их, затем предоставлять их специальной процедуре, аналогичной SkipWhite, которая пропускает их только в определенных «допустимых» местах.

Вот эта процедура:

```
{--------------------------------------------------------------}
{ Skip a CRLF }
procedure Fin;
begin
if Look = CR then GetChar;
if Look = LF then GetChar;
end;
{--------------------------------------------------------------}
```

Теперь добавьте два вызова Fin в процедуру Block следующим образом:

```
{--------------------------------------------------------------}
{ Recognize and Translate a Statement Block }
procedure Block(L: string);
begin
while not(Look in ['e', 'l', 'u']) do begin
Fin;
case Look of
```

```
'i': DoIf(L);
'w': DoWhile;
'p': DoLoop;
'r': DoRepeat;
'f': DoFor;
'd': DoDo;
'b': DoBreak(L);
else Other;
end;
Fin;
end;
end;
_
```

Теперь вы обнаружите, что можете использовать многострочные «программы». Единственное ограничение в том, что вы не можете отделять токены IF или WHILE от их предикатов.

Теперь мы готовы включить операторы присваивания. Просто замените вызов Other в процедуре Block на вызов Assignment и добавьте следующую процедуру, скопированную из одной нашей более ранней программы. Обратите внимание, что сейчас Assignment вызывает BoolExpression, поэтому мы можем присваивать логические переменные.

```
_
 Parse and Translate an Assignment Statement
procedure Assignment;
var Name: char;
begin
Name := GetName;
Match('=');
BoolExpression;
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE D0,(A0)');
end;
_
```

С этими изменениями у вас теперь должна быть возможность писать сносные, реалистично выглядящие программы, подчиненные только нашему ограничению односимвольными токенами. Первоначально я также намеревался избавить вас и от этого ограничения. Однако, это потребует довольно больших изменений того, что мы сделали к этому моменту. Нам нужен настоящий лексический анализатор и это требует некоторых структурных изменений. Это небольшие изменения, которые потребуют чтобы мы выбросили все, что мы сделали к этому времени... при желании это может быть сделано в действительности с минимальными изменениями. Но необходимо такое желание.

Эта глава и так получилась довольно длинной и она содержит довольно тяжелый материал, поэтому я решил оставить этот шаг до следующего раза, чтобы у вас было немного времени усвоить то, что мы сделали и вы были готовы начать на свежую голову.

В следующей главе, мы построим лексический анализатор и устраним односимвольный барьер раз и навсегда. Мы также напишем наш первый законченный компилятор, основанный на том, что мы сделали на этом уроке. Увидимся.

Лексический анализ

ВВЕДЕНИЕ

В последней главе я оставил вас с компилятором который должен почти работать, за исключением того, что мы все еще ограничены односимвольными токенами. Цель этого урока состоит в том, чтобы избавиться от этого ограничения раз и навсегда. Это означает, что мы должны иметь дело с концепцией лексического анализатора (сканера).

Возможно я должен упомянуть, почему нам вообще нужен лексический анализатор... в конце концов до настоящего времени мы были способны хорошо справляться и без него даже когда мы предусмотрели многосимвольные токены.

Единственная причина, на самом деле, имеет отношение к ключевым словам. Это факт компьютерной жизни, что синтаксис ключевого слова имеет ту же самую форму, что и синтаксис любого другого идентификатора. Мы не можем сказать пока не получим полное слово действительно ли это ключевое слово. К примеру переменная IFILE и ключевое слово IF выглядят просто одинаковыми до тех пор, пока вы не получите третий символ. В примерах до настоящего времени мы были всегда способны принять решение, основанное на первом символе токена, но это больше невозможно когда присутствуют ключевые слова. Нам необходимо знать, что данная строка является ключевым словом до того, как мы начнем ее обрабатывать. И именно поэтому нам нужен сканер.

На последнем уроке я также пообещал, что мы могли бы предусмотреть нормальные токены без глобальных изменений того, что мы уже сделали. Я не солгал... мы можем, как вы увидите позднее. Но каждый раз, когда я намеревался встроить эти элементы в синтаксический анализатор, который мы уже построили, у меня возникали плохие чувства в отношении их. Все это слишком походило на временную меру. В конце концов я выяснил причину проблемы: я установил программу лексического анализа не объяснив вам вначале все о лексическом анализе, и какие есть альтернативы. До настоящего времени я старательно избегал давать вам много теории и, конечно, альтернативные варианты. Я обычно не воспринимаю хорошо учебники которые дают двадцать пять различных способов сделать что-то, но никаких сведений о том, какой способ лучше всего вам подходит. Я попытался избежать этой ловушки, просто показав вам один способ, который работает.

Но это важная область. Хотя лексический анализатор едва ли является наиболее захватывающей частью компилятора он часто имеет наиболее глубокое влияние на общее восприятие языка так как эта часть наиболее близка пользователю. Я придумал специфическую структуру сканера, который будет использоваться с KISS. Она соответствует восприятию, которое я хочу от этого языка. Но она может совсем не работать для языка, который придумаете вы, поэтому в этом единственном случае я чувствую, что вам важно знать ваши возможности.

Поэтому я собираюсь снова отклониться от своего обычного распорядка. На этом уроке мы заберемся гораздо глубже, чем обычно, в базовую теорию языков и грамматик. Я также буду говорить о других областях кроме компиляторов в которых лексических анализ играет важную роль. В заключение я покажу вам некоторые альтернативы для структуры лексического анализатора. Тогда и только тогда мы возвратимся к нашему синтаксическому анализатору из последней главы. Потерпите... я думаю вы найдете, что это стоит ожидания. Фактически, так как сканеры имеют множество применений вне компиляторов, вы сможете легко убедиться, что это будет наиболее полезный для вас урок.

ЛЕКСИЧЕСКИЙ АНАЛИЗ

Лексический анализ – это процесс сканирования потока входных символов и разделения его на строки, называемые лексемами. Большинство книг по компиляторам начинаются с этого и посвящают несколько глав обсуждению различных методов построения сканеров. Такой подход имеет свое место, но, как вы уже видели, существуют множество вещей, которые вы можете сделать даже никогда не обращавшись к этому вопросу, и, фактически, сканер, который мы здесь закончим, не очень будет напоминать то,

что эти тексты описывают. Причина? Теория компиляторов и, следовательно, программы следующие из нее, должны работать с большинством общих правил синтаксического анализа. Мы же не делаем этого. В реальном мире возможно определить синтаксис языка таким образом, что будет достаточно довольно простого сканера. И как всегда KISS – наш девиз.

Как правило, лексический анализатор создается как отдельная часть компилятора, так что синтаксический анализатор по существу видит только поток входных лексем. Теоретически нет необходимости отделять эту функцию от остальной части синтаксического анализатора. Имеется только один набор синтаксических уравнений, который определяет весь язык, поэтому теоретически мы могли бы написать весь анализатор в одном модуле.

Зачем необходимо разделение? Ответ имеет и теоретическую и практическую основы.

В 1956 Ноам Хомский определил «Иерархию Хомского» для грамматик. Вот они:

Тип 0. Неограниченные (например Английский язык)

Тип 1. Контекстно-зависимые

Тип 2. Контекстно-свободные

Тип 3. Регулярные.

Некоторые характеристики типичных языков программирования (особенно старых, таких как Фортран) относят их к Типу 1, но большая часть всех современных языков программирования может быть описана с использованием только двух последних типов и с ними мы и будем здесь работать.

Хорошая сторона этих двух типов в том, что существуют очень специфические пути для их анализа. Было показано, что любая регулярная грамматика может быть анализирована с использованием частной формы абстрактной машины, называемой конечным автоматом. Мы уже реализовывали конечные автоматы в некоторых их наших распознающих программ.

Аналогично грамматики Типа 2 (контекстно-свободные) всегда могут быть анализированы с использованием магазинного автомата (конечный автомат, дополненный стеком). Мы также реализовывали эти машины. Вместо реализации явного стека для выполнения работы мы положились на встроенный стек связанный с рекурсивным кодированием и это фактически является предочтительным способом для нисходящего синтаксического анализа.

Случается что в реальных, практических грамматиках части, которые квалифицируются как регулярные выражения, имеют склонность быть низкоуровневыми частями, как определение идентификатора:

<ident> ::= <letter> [<letter> | <digit>]*

Так как требуется различные виды абстрактных машин для анализа этих двух типов грамматик, есть смысл отделить эти низкоуровневые функции в отдельный модуль, лексический анализатор, который строится на идее конечного автомата. Идея состоит в том, чтобы использовать самый простой метод синтаксического анализа, необходимый для работы.

Имеется другая, более практическая причина для отделения сканера от синтаксического анализатора. Мы хотим думать о входном исходном файле как потоке символов, которые мы обрабатываем справа налево без возвратов. На практике это невозможно. Почти каждый язык имеет некоторые ключевые слова типа IF, WHILE и END. Как я упомянул ранее, в действительности мы не можем знать является ли данная строка ключевым словом до тех пор пока мы не достигнем ее конца, что определено пробелом или другим разделителем. Так что мы должны хранить строку достаточно долго для того, чтобы выяснить имеем мы ключевое слово или нет. Это ограниченная форма перебора с возвратом.

Поэтому структура стандартного компилятора включает разбиение функций низкоуровневого и высокоуровневого синтаксического анализа. Лексический анализатор работает на символьном уровне собирая символы в строки и т.п., и передавая их синтаксическому анализатору как неделимые лексемы. Также считается нормальным позволить сканеру выполнять работу по идентификации ключевых слов.

КОНЕЧНЫЕ АВТОМАТЫ И АЛЬТЕРНАТИВЫ

Я упомянул, что регулярные выражения могут анализироваться с использованием конечного автомата. В большинстве книг по компиляторам а также в большинстве компиляторов, вы обнаружите, что это применяется буквально. Обычно они имеют настоящую реализацию конечного автомата с целыми числами, используемыми для определения текущего состояния и таблицей действий, выполняемых для каждой комбинации текущего состояния и входного символа. Если вы пишите «front end» для компилятора, используя популярные Unix инструменты LEX и YACC, это то, что вы получите. Выход LEX – конечый автомат, реализованный на C плюс таблица действий, соответствующая входной грамматике данной LEX. Вывод YACC аналогичен... исскуственный таблично-управляемый синтаксический анализатор плюс таблица, соответствующая синтаксису языка.

Однако это не единственный вариант. В наших предыдущих главах вы много раз видели, что возможно реализовать синтаксические анализаторы специально не имея дела с таблицами, стеками и переменными состояния. Фактически в пятой главе я предупредил вас, что если вы считает себя нуждающимся в этих вещах, возможно вы делаете что-то неправильно и не используете возможности Паскаля. Существует в основном два способа определить состояние конечного автомата: явно, с номером или кодом состояния и неявно, просто на основании того факта, что я нахожусь в каком-то определенном месте кода (если сегодня вторник, то это должно быть Бельгия). Ранее мы полагались в основном на неявные методы, и я думаю вы согласитесь, что они работают здесь хорошо.

На практике может быть даже не обязательно иметь четко определенный лексический анализатор. Это не первый наш опыт работы с многосимвольными токенами. В третьей главе мы расширили наш синтаксический анализатор для их поддержки и нам даже не был нужен лексический анализатор. Причиной было то, что в узком контексте мы всегда могли сказать просто рассматривая единственный предсказывающий символ, имеем ли мы дело с цифрой, переменной или оператором. В действительности мы построили распределенный лексический анализатор, используя процедуры GetName и GetNum.

Имея ключевые слов мы не можем больше знать с чем мы имеем дело до тех пор, пока весь токен не будет прочитан. Это ведет нас к более локализованному сканеру, хотя, как вы увидите, идея распределенного сканера все же имеет свои достоинства.

ЭКСПЕРИМЕНТЫ ПО СКАНИРОВАНИЮ

Прежде чем возвратиться к нашему компилятору, было бы полезно немного поэкспериментировать с общими понятиями.

Давайте начнем с двух определений, наиболее часто встречающихся в настоящих языках программирования:

<ident> ::= <letter> [<letter> | <digit>]*

<number ::= [<digit>]+

(Не забудьте, что "*" указывает на ноль или более повторений условия в квадратных скобках, а "+" на одно и более.)

Мы уже работали с подобными элементами в третьей главе. Давайте начнем (как обычно) с пустого Cradle. Не удивительно, что нам понадобится новая процедура распознавания:

```
{--------------------------------------------------------------}
{ Recognize an Alphanumeric Character }
function IsAlNum(c: char): boolean;
begin
IsAlNum := IsAlpha(c) or IsDigit(c);
```

```
end;
–
```

Используя ее, давайте напишем следующие две подпрограммы, которые очень похожи на те, которые мы использовали раньше:

```
–
 Get an Identifier
function GetName: string;
var x: string[8];
begin
x := '';
if not IsAlpha(Look) then Expected('Name');
while IsAlNum(Look) do begin
x := x + UpCase(Look);
GetChar;
end;
GetName := x;
end;
–
 Get a Number
function GetNum: string;
var x: string[16];
begin
x := '';
if not IsDigit(Look) then Expected('Integer');
while IsDigit(Look) do begin
x := x + Look;
GetChar;
end;
GetNum := x;
end;
–
```

(Заметьте, что эта версия GetNum возвращает строку, а не целое число, как прежде).

Вы можете легко проверить что эти подпрограммы работают, вызвав их из основной программы:

```
WriteLn(GetName);
```

Эта программа выведет любое допустимое набранное имя (максимум восемь знаков, потому что мы так сказали GetName). Она отвергнет что-либо другое.

Аналогично проверьте другую подпрограмму.

ПРОБЕЛ

Раньше мы также работали с вложенными пробелами, используя две подпрограммы IsWhite и SkipWhite. Удостоверьтесь, что эти подпрограммы есть в вашей текущей версии Cradle и добавьте строку:

```
SkipWhite;
```

в конец GetName и GetNum.

Теперь давайте определим новую процедуру:

```
–
 Lexical Scanner
Function Scan: string;
begin
```

```
if IsAlpha(Look) then
Scan := GetName
else if IsDigit(Look) then
Scan := GetNum
else begin
Scan := Look;
GetChar;
end;
SkipWhite;
end;
_
```

Мы можем вызвать ее из новой основной программы:

```
_
 Main Program
begin
Init;
repeat
Token := Scan;
writeln(Token);
until Token = CR;
end.
_
```

(Вы должны добавить описание строки Token в начало программы. Сделайте ее любой удобной длины, скажем 16 символов).

Теперь запустите программу. Заметьте, что входная строка действительно разделяется на отдельные токены.

КОНЕЧНЫЕ АВТОМАТЫ

Подпрограмма анализа типа GetName действительно реализует конечный автомат. Состояние неявно в текущей позиции в коде. Очень полезным приемом для визуализации того, что происходит, является синтаксическая диаграмма или «railroad-track» диаграмма. Немного трудно нарисовать их в этой среде, поэтому я буду использовать их очень экономно, но фигура ниже должна дать вам идею:

Как вы можете видеть, эта диаграмма показывает логические потоки по мере чтения символов. Начинается все, конечно, с состояния «start» и заканчивается когда найден символ, отличный от алфавитно-цифрового. Если первый символ не буква, происходит ошибка. Иначе автомат продолжит выполнение цикла до тех пор, пока не будет найден конечный разделитель.

Заметьте, что в любой точке потока наша позиция полностью зависит от предыдущей истории входных символов. В этой точке предпринимаемые действия зависят только от текущего состояния плюс текущий входной символ. Это и есть то, что образует конечный автомат.

Из-за сложностей представления «railroad-track» диаграмм в этой среде я буду продолжать придерживаться с этого времени синтаксических уравнений. Но я настоятельно рекомендую вам диаграммы для всего, что включает синтаксический анализ. После небольшой практики вы можете начать видеть, как написать синтаксический анализатор непосредственно из диаграммы. Параллельные пути кодируются в контролирующие действия (с помощью операторов IF или CASE), последовательные пути – в последовательные вызовы. Это почти как работа по схеме.

Мы даже не обсудили SkipWhite, которая была представлена раньше, но это также

простой конечный автомат, как и GetNum. Так же как и их родительская процедура Scan. Маленькие автоматы образуют большие автоматы.

Интересная вещь, на которую я хотел бы чтобы вы обратили внимание это то, как безболезненно такой неявный подход создает эти конечные автоматы. Я лично предпочитаю его таблично-управляемому методу. Он также получает маленькие, компактные и быстрые сканеры.

НОВЫЕ СТРОКИ

Продвигаясь прямо вперед, давайте модифицируем наш сканер для поддержки более чем одной строки. Как я упомянул последний раз, наиболее простой способ сделать это – просто обработать символы новой строки, возврат каретки и перевод строки, как незаполненное пространство. Фактически это способ, используемый подпрограммой iswhite из стандартной библиотеки C. Прежде мы не этого делали. Я хотел бы сделать это теперь, чтобы вы могли почувствовать результат.

Чтобы сделать это просто измените единственную выполнимую строку в IsWhite:

```
IsWhite := c in [' ', TAB, CR, LF];
```

Мы должны дать основной программы новое условие останова, так как она никогда не увидит CR. Давайте просто используем:

```
until Token = '.';
```

ОК, откомпилируйте эту программу и запустите ее. Попробуйте пару строк, завершаемых точкой. Я использовал:

```
now is the time
for all good men.
```

Эй, что случилось? Когда я набрал это, я не получил последний токен, точку. Программа не остановилась. Более того, когда я нажал клавишу 'enter' несколько раз, я все равно не получил точку.

Если вы все еще не можете выбраться из вашей программы, вы обнаружите, что набор точки в новой строке прервет ее.

Что здесь происходит? Ответ в том, что мы зависаем в SkipWhite. Короткий осмотр этой подпрограммы покажет, что пока мы печатаем пустые строки, мы просто продолжаем выполнение цикла. После того, как SkipWhite встречает LF, он пытается выполнить GetChar. Но так как входной буфер теперь пуст, оператор чтения в GetChar настаивает на наличии другой строки. Процедура Scan получает завершающую точку, все правильно, но она вызывает SkipWhite и SkipWhite не возвращается до тех пор, пока не получит непустую строку.

Такое поведение не настолько плохое, как кажется. В настоящем компиляторе мы читали бы символы из входного файла вместо консоли и пока мы имеем какую-то процедуру для работы с концом файла, все получится ОК. Но для чтения данных с консоли такое поведение слишком причудливое. Суть в том, что соглашение C/Unix просто не совместимо со структурой нашего анализатора, который запрашивает предсказывающий символ. Код, который мастера из Bell реализовали, не использует это соглашение, поэтому они нуждаются в 'ungetc'.

ОК, давайте исправим проблему. Чтобы сделать это, мы должны возвратиться к старому определению IsWhite (удалите символы CR и LF) и используйте процедуру Fin, которую я представил в последний раз. Если ее нет в вашей текущей версии Cradle, поместите ее там.

Также измените основную программу следующим обраазом:

```
_
 Main Program
begin
Init;
```

```
repeat
Token := Scan;
writeln(Token);
if Token = CR then Fin;
until Token = '.';
end.
–
```

Обратите внимание на «охраняющую» проверку, предшествующую вызову Fin. Это то, что заставляет все это работать, и проверяет, то мы не пытаемся прочитать строку дальше.

Сейчас испытайте этот код. Я думаю он понравится вам больше.

Если вы обратитесь к коду, который мы написали в последней главе, вы обнаружите, что я расставил вызовы Fin по всему коду, где прерывание строки было бы уместным. Это одна из тех областей, которые действительно влияют на восприятие, о котором я упомянул. В этой точке я должен убедить вас поэкспериментировать с различными способами организациями и посмотреть, как вам это понравится. Если вы хотите, чтобы ваш язык был по настоящему свободного стиля, тогда новые строки должны быть прозрачны. В этом случае наилучшим подходом было бы поместить следующие строки в начале Scan:

```
while Look = CR do
Fin;
```

Если, с другой стороны, вам нужен строчно-ориентированный язык подобный Ассемблеру, BASIC или FORTRAN (или даже Ada... заметьте, что он имеет комментарии, завершаемые новой строкой), тогда вам необходимо, чтобы Scan возвращал CR как токены. Он также должен съедать завершающие LF. Лучший способ сделать – использовать эту строку в самом начале Scan:

```
if Look = LF then Fin;
```

Для других соглашений вы будете должны использовать другие способы организации. В моем примере на последнем уроке я разрешил новые строки только в определенных местах, поэтому я занял какое-то промежуточное положение. В остальных частях этих занятий я буду выбирать такие способы обработки новых строк какие мне понравятся, но я хочу, чтобы вы знали, как выбрать для себя другой путь.

ОПЕРАТОРЫ

Мы могли бы сейчас остановиться и иметь в своем распоряжении довольно полезный сканер. В тех фрагментах KISS, которые мы построили, единственными токенами, состоящими из нескольких символов, являются идентификаторы и числа. Все операторы были односимвольными. Единственное исключение, которое я могу придумать – это операторы отношений «<=», «>=» и «<>», но они могут быть обработаны как особые случаи.

Однако другие языки имеют многосимвольные операторы такие как «:=» в Паскале или «++» и «>>» в C. Хотя пока нам и не нужны многосимвольные операторы, было бы хорошо знать как получить их в случае необходимости.

Само собой разумеется, что мы можем обрабатывать операторы точно таким же способом, что и другие токены. Давайте начнем с подпрограммы распознавания:

```
–
 Recognize Any Operator
function IsOp(c: char): boolean;
begin
IsOp := c in ['+', '-', '*', '/', '&amp;lt;', '&amp;gt;', ':', '='];
end;
–
```

Важно заметить, что мы не должны включать в этот список каждый возможный оператор. К примеру круглые скобки не включены, так же как и завершающая точка. Текущая версия Scan и так хорошо поддерживает односимвольные операторы. Список выше включает только те символы, которые могут появиться в многосимвольных операторах. (Для конкретных языков список конечно всегда может быть отредактирован).

Теперь давайте изменим Scan следующим образом:

```
{
 Lexical Scanner
}
Function Scan: string;
begin
while Look = CR do
Fin;
if IsAlpha(Look) then
Scan := GetName
else if IsDigit(Look) then
Scan := GetNum
else if IsOp(Look) then
Scan := GetOp
else begin
Scan := Look;
GetChar;
end;
SkipWhite;
end;
{
```

Теперь испытайте программу. Вы убедитесь, что любые фрагменты кода, которые вы захотите бросить в нее будут аккуратно разложены на индивидуальные токены.

СПИСКИ, ЗАПЯТЫЕ И КОМАНДНЫЕ СТРОКИ.

Прежде чем возвратиться к основной цели нашего обучения, я хотел бы немного выступить.

Сколько раз вы работали с программой или операционной системой, которая имела жесткие правила того, как вы должны разделять элементы в списке? (Попробую, последний раз вы использовали MS DOS!). Некоторые программы требуют пробелов как разделителей, некоторые требуют запятые. Хуже всего, что некоторые требуют и того и другого в разных местах. Большинство довольно неумолимы к нарушениям их правил.

Я думаю, это непростительно. Слишком просто написать синтаксически анализатор, который поддерживает и пробелы и запятые гибким способом. Рассмотрите следующую процедуру:

```
{
 Skip Over a Comma
}
procedure SkipComma;
begin
SkipWhite;
if Look = ',' then begin
GetChar;
SkipWhite;
end;
end;
{
```

Эта процедура из восьми строк пропустит разделитель, состоящий из любого числа (включая ноль) пробелов, с нулем или одной запятой, вложенной в строку.

Временно измените вызов SkipWhite в Scan на вызов SkipComma и попробуйте ввести какие-нибудь списки. Хорошо работает, да? Разве вы не хотите, чтобы больше создателей программ знало о SkipComma?

К слову сказать, я обнаружил, что добавление эквивалента SkipComma в мою программу на ассемблере для Z80 заняло всего шесть дополнительных байт кода. Даже на 64K машинах это не слишком большая цена за дружелюбие к пользователю.

Я думаю вы можете видеть к чему я клоню. Даже если вы в своей жизни не написали ни одной строчки кода для компилятора, в каждой программе существуют места, где вы можете использовать понятие синтаксического анализа. Любая программа, которая обрабатывает командные строки, нуждается в нем. Фактически, если вы подумаете немного об этом, вы придете к заключению, что всякий раз, когда вы пишете программу, обрабатывающую ввод пользователя, вы определяете язык. Люди общаются с помощью языков и неявный синтаксис в вашей программе определяет этот язык. Настоящий вопрос: вы собираетесь определять его преднамеренно и явно, или просто позволите существовать независимо от того, как программа завершает синтаксический анализ?

Я утверждаю, что у вас будет лучший, более дружественный интерфейс если вы потратите время на то, чтобы определить синтаксис явно. Запишите синтаксические уравнения или нарисуйте «railroad-track» диаграммы и закодируйте синтаксический анализатор используя методы, которые я показал вам здесь. Вы получите более хорошую программу и ее будет проще писать, в придачу.

СТАНОВИТСЯ ИНТЕРЕСНЕЙ

Хорошо, сейчас мы имеем довольно хороший лексический анализатор, который разбивает входной поток на лексемы. Мы могли бы использовать его как есть и иметь полезный компилятор. Но есть некоторые другие аспекты лексического анализа, которые мы должны охватить.

Особенно следует рассмотреть (вздрогните) эффективность. Помните, когда мы работали с односимвольными токенами, каждой проверкой было сравнение одного символа Look с байтовой константой. Мы также использовали в основном оператор Case.

С многосимвольными лексемами, возвращаемыми Scan, все эти проверки становятся сравнением строк. Гораздо медленнее. И не только медленнее но и неудобней, так как в Паскале не существут строкового эквивалента оператора Case. Особенно расточительным кажется проверять то что состоит из одного символа... "=", "+" и другие операторы... используя сравнение строк.

Сравнение строк не является невозможным. Рон Кейн использовал этот подход при написании Small C. Так как мы придерживаемся принципа KISS мы были бы оправданы согласившись с этим подходом. Но тогда я не смог бы рассказать вам об одном из ключевых методов, используемых в «настоящих» компиляторах.

Вы должны запомнить: лексический анализатор будет вызываться часто! Фактически один раз для каждой лексемы во всей исходной программе. Эксперименты показали, что средний компилятор тратит где-то от 20 до 40 процентов своего времени на подпрограммах лексического анализа. Если существовало когда-либо место, где эффективность заслуживает пристального рассмотрения, то это оно.

По этой причине большинство создателей компиляторов заставляют лексический анализатор выполнять немного больше работы, «токенизируя» входной поток. Идея состоит в том, чтобы сравнивать каждую лексему со списком допустимых ключевых слов и операторов и возвращать уникальный код для каждой распознанной. В случае обычного имени переменной или числа мы просто возвращаем код, который говорит, к какому типу

лексем они относятся и сохраняем где-нибудь текущую строку.

Первое, что нам нужно – это способ идентификации ключевых слов. Мы всегда можем сделать это с помощью последовательных проверок IF, но несомненно было бы хорошо, если бы мы имели универсальную подпрограмму, которая могла бы сравнивать данную строку с таблицей ключевых слов. (Между прочим, позднее нам понадобится такая же подпрограмма для работы с таблицей идентификаторов). Это обычно выявляет проблему Паскаля, потому что стандартный Паскаль не имеет массивов переменной длины. Это настоящая головная боль – обьявлять различные подпрограммы поиска для каждой таблицы. Стандартный Паскаль также не позволяет инициализировать массивы, поэтому вам придется видеть код типа:

```
Table[1] := 'IF';
Table[2] := 'ELSE';
.
.
Table[n] := 'END';
```

что может получиться довольно длинным если есть много ключевых слов.

К счастью Turbo Pascal 4.0 имеет расширения, которые устраняют обе эти проблемы. Массивы-константы могут быть обьявлены с использованием средства TP «типизированные константы» а переменные размерности могут быть поддержаны с помощью Си-подобных расширений для указателей.

Сначала, измените ваши объявления подобным образом:

```
_
 Type Declarations
type Symbol = string[8];
SymTab = array[1..1000] of Symbol;
TabPtr = ^SymTab;
_
```

(Размерность, использованная в SymTab не настоящая... память не распределяется непосредственно этим объявлением, а размерность должна быть только «достаточно большой»)

Затем, сразу после этих объявлений, добавьте следующее:

```
_
 Definition of Keywords and Token Types
const KWlist: array [1..4] of Symbol =
('IF', 'ELSE', 'ENDIF', 'END');
_
```

Затем, вставьте следующую новую функцию:

```
_
 Table Lookup
 If the input string matches a table entry, return the entry
index. If not, return a zero.
function Lookup(T: TabPtr; s: string; n: integer): integer;
var i: integer;
found: boolean;
begin
found := false;
i := n;
while (i &gt; 0) and not found do
if s = T^[i] then
```

```
found := true
else
dec(i);
Lookup := i;
end;
–
```

Чтобы проверить ее вы можете временно изменить основную программу следующим образом:

```
–
 Main Program
begin
ReadLn(Token);
WriteLn(Lookup(Addr(KWList), Token, 4));
end.
–
```

Обратите внимание как вызывается Lookup: функция Addr устанавливает указатель на KWList, который передается в Lookup.

ОК, испытайте ее. Так как здесь мы пропускаем Scan, для получения соответствия вы должны набирать ключевые слова в верхнем регистре.

Теперь, когда мы можем распознавать ключевые слова, далее необходимо договориться о возвращаемых для них кодах.

Итак, какие кода мы должны возвращать? В действительности есть только два приемлемых варианта. Это похоже на идеальное применения перечислимого типа Паскаля. К примеру, вы можете определить что-то типа

SymType = (IfSym, ElseSym, EndifSym, EndSym, Ident, Number, Operator);

и договориться возвращать переменную этого типа. Давайте попробуем это. Вставьте строку выше в описание типов.

Теперь добавьте два описания переменных:

Token: Symtype; Current Token

Value: String[16]; String Token of Look

Измените сканер так:

```
–
 Lexical Scanner
procedure Scan;
var k: integer;
begin
while Look = CR do
Fin;
if IsAlpha(Look) then begin
Value := GetName;
k := Lookup(Addr(KWlist), Value, 4);
if k = 0 then
Token := Ident
else
Token := SymType(k – 1);
end
else if IsDigit(Look) then begin
Value := GetNum;
Token := Number;
end
else if IsOp(Look) then begin
```

```
Value := GetOp;
Token := Operator;
end
else begin
Value := Look;
Token := Operator;
GetChar;
end;
SkipWhite;
end;
_
```

(Заметьте, что Scan сейчас стала процедурой а не функцией).
Наконец, измените основную программу:

```
_
 Main Program
begin
Init;
repeat
Scan;
case Token of
Ident: write('Ident ');
Number: Write('Number ');
Operator: Write('Operator ');
IfSym, ElseSym, EndifSym, EndSym: Write('Keyword ');
end;
Writeln(Value);
until Token = EndSym;
end.
_
```

Мы заменили строку Token, используемую раньше, на перечислимый тип. Scan возвращает тип в переменной Token и возвращает саму строку в новой переменной Value.

ОК, откомпилируйте программу и погоняйте ее. Если все работает, вы должны увидеть, что теперь мы распознаем ключевые слова.

Теперь у нас все работает правильно, и было легко сгенерировать это из того, что мы имели раньше. Однако, она все равно кажется мне немного «перегруженной». Мы можем ее немного упростить, позволив GetName, GetNum, GetOp и Scan работать с глобальными переменными Token и Value, вследствие этого удаляя их локальные копии. Кажется немного умней было бы переместить просмотр таблицы в GetName. Тогда новая форма для этих четырех процедур будет такой:

```
_
 Get an Identifier
procedure GetName;
var k: integer;
begin
Value := '';
if not IsAlpha(Look) then Expected('Name');
while IsAlNum(Look) do begin
Value := Value + UpCase(Look);
GetChar;
end;
k := Lookup(Addr(KWlist), Value, 4);
if k = 0 then
```

```
Token := Ident
else
Token := SymType(k-1);
end;
–
 Get a Number
procedure GetNum;
begin
Value := '';
if not IsDigit(Look) then Expected('Integer');
while IsDigit(Look) do begin
Value := Value + Look;
GetChar;
end;
Token := Number;
end;
–
 Get an Operator
procedure GetOp;
begin
Value := '';
if not IsOp(Look) then Expected('Operator');
while IsOp(Look) do begin
Value := Value + Look;
GetChar;
end;
Token := Operator;
end;
–
 Lexical Scanner
procedure Scan;
var k: integer;
begin
while Look = CR do
Fin;
if IsAlpha(Look) then
GetName
else if IsDigit(Look) then
GetNum
else if IsOp(Look) then
GetOp
else begin
Value := Look;
Token := Operator;
GetChar;
end;
SkipWhite;
end;
–
```

ВОЗВРАЩЕНИЕ СИМВОЛА

По существу, все сканеры, которые я когда-либо видел и которые написаны на Паскале, использовали механизм перечислимых типов, который я только что описал. Это конечно работающий механизм, но он не кажется мне самым простым подходом.

Прежде всего, список возможных типов символов может получиться довольно длинным. Здесь я использовал только один символ «Operator» для обозначения всех операторов, но я видел другие проекты, в которых фактически возвращаются различные кода для каждого.

Существует, конечно, другой простой тип, который может быть возвращен как код: символ. Вместо возвращения значения «Operator» для знака "+", что неправильного в том, чтобы просто возвращать сам символ? Символ – такая же хорошая переменная для кодирования различных типов лексем, она легко может быть использована в операторах Case, и это гораздо проще набрать. Что может быть проще?

Кроме того, мы уже имели опыт с идеей кодировать ключевые слова как одиночные символы. Наши предыдущие программы уже написаны таким способом, так что использование этого метода минимизирует изменения того, что мы уже сделали.

Некоторые из вас могут почувствовать, что идея с возвращение символьных кодов слишком детская. Я должен допустить, что она становится немного неуклюжей для операторов типа «<=». Если вы хотите остаться с перечислимыми типами, хорошо. Для остальных я хотел бы показать как изменить то, что мы сделали выше, для поддержки такого подхода.

Во-первых, сейчас вы можете удалить объявление типа SymType... он нам больше не понадобится. И вы можете изменить тип Token в char.

Затем, чтобы заменить SymType, добавьте следующую константу:

const KWcode: string[5] = 'xilee';

(Я буду кодировать все идентификаторы одиночным символом 'x').

Наконец измените Scan и его родственников следующим образом:

```
{--------------------------------------------------------------}
{ Get an Identifier }
procedure GetName;
begin
Value := '';
if not IsAlpha(Look) then Expected('Name');
while IsAlNum(Look) do begin
Value := Value + UpCase(Look);
GetChar;
end;
Token := KWcode[Lookup(Addr(KWlist), Value, 4) + 1];
end;
{--------------------------------------------------------------}
{ Get a Number }
procedure GetNum;
begin
Value := '';
if not IsDigit(Look) then Expected('Integer');
while IsDigit(Look) do begin
Value := Value + Look;
GetChar;
end;
Token := '#';
end;
{--------------------------------------------------------------}
{ Get an Operator }
procedure GetOp;
begin
Value := '';
if not IsOp(Look) then Expected('Operator');
while IsOp(Look) do begin
```

```
Value := Value + Look;
GetChar;
end;
if Length(Value) = 1 then
Token := Value[1]
else
Token := '?';
end;
–
 Lexical Scanner
procedure Scan;
var k: integer;
begin
while Look = CR do
Fin;
if IsAlpha(Look) then
GetName
else if IsDigit(Look) then
GetNum
else if IsOp(Look) then begin
GetOp
else begin
Value := Look;
Token := '?';
GetChar;
end;
SkipWhite;
end;
–
 Main Program
begin
Init;
repeat
Scan;
case Token of
'x': write('Ident ');
'#': Write('Number ');
'i', 'l', 'e': Write('Keyword ');
else Write('Operator ');
end;
Writeln(Value);
until Value = 'END';
end.
–
```

Эта программа должна работать также как и предыдущая версия. Небольшое различие в структуре, может быть, но она кажется мне более простой.

РАСПРЕДЕЛЕННЫЕ СКАНЕРЫ ПРОТИВ ЦЕНТРАЛИЗОВАННЫХ

Структура лексического анализатора, которую я только что вам показал, весьма стандартна и примерно 99% всех компиляторов используют что-то очень близкое к ней. Это, однако, не единственно возможная структура, или даже не всегда самая лучшая.

Проблема со стандартным подходом состоит в том, что сканер не имеет никаких сведений о контексте. Например, он не может различить оператор присваивания "=" и оператор отношения "=" (возможно именно поэтому и C и Паскаль используют для них

различные строки). Все, что сканер может сделать, это передать оператор синтаксическому анализатору, который может точно сказать исходя из контекста, какой это оператор. Точно так же, ключевое слово «IF» не может быть посредине арифметического выражения, но если ему случится оказаться там, сканер не увидит в этом никакой проблемы и возвратит его синтаксическому анализатору, правильно закодировав как «IF».

С таким подходом, мы в действительности не используем всю информацию, имеющуюся в нашем распоряжении. В середине выражения, например, синтаксический анализатор «знает», что нет нужды искать ключевое слово, но он не имеет никакой возможности сказать это сканеру. Так что сканер продолжает делать это. Это, конечно, замедляет компиляцию.

В настоящих компиляторах проектировщики часто принимают меры для передачи подробной информации между сканером и парсером, только чтобы избежать такого рода проблем. Но это может быть неуклюже и, конечно, уничтожит часть модульности в структуре компилятора.

Альтернативой является поиск какого-то способа для использования контекстной информации, которая исходит из знания того, где мы находимся в синтаксическом анализаторе. Это возвращает нас обратно к понятию распределенного сканера, в котором различные части сканера вызываются в зависимости от контекста.

В языке KISS, как и большинстве языков, ключевые слова появляются только в начале утверждения. В таких местах, как выражения они запрещены. Также, с одним небольшим исключением (многосимвольные операторы отношений), которое легко обрабатывается, все операторы односимвольны, что означает, что нам совсем не нужен GetOp.

Так что, оказывается, даже с многосимвольными токенами мы все еще можем всегда точно определить вид лексемы исходя из текущего предсказывающего символа, исключая самое начало утверждения.

Даже в этой точке, единственным видом лексемы, который мы можем принять, является идентификатор. Нам необходимо только определить, является ли этот идентификатор ключевым словом или левой частью оператора присваивания.

Тогда мы заканчиваем все еще нуждаясь только в GetName и GetNum, которые используются так же, как мы использовали их в ранних главах.

Сначала вам может показаться, что это шаг назад и довольно примитивный способ. Фактически же, это усовершенствование классического сканера, так как мы используем подпрограммы сканирования только там, где они действительно нужны. В тех местах, где ключевые слова не разрешены, мы не замедляем компиляцию, ища их.

ОБЪЕДИНЕНИЕ СКАНЕРА И ПАРСЕРА

Теперь, когда мы охватили всю теорию и общие аспекты лексического анализа, я наконец готов подкрепит свое заявление о том, что мы можем приспособить многосимвольные токены с минимальными изменениями в нашей предыдущей работе. Для краткости и простоты я ограничу сам себя подмножеством того, что мы сделали ранее: я разрешу только одну управляющую конструкцию (IF) и никаких булевых выражений. Этого достаточно для демонстрации синтаксического анализа и ключевых слов и выражений. Расширение до полного набора конструкций должно быть довольно очевидно из того, что мы уже сделали.

Все элементы программы для синтаксического анализа этого подмножества с использованием односимвольных токенов уже существуют в наших предыдущих программах. Я построил ее осторожно скопировав эти файлы, но я не посмею попробовать провести вас через этот процесс. Вместо этого, во избежание беспорядка, вся программа показана ниже:

–

```
program KISS;
{
 Constant Declarations
}
const TAB = ^I;
CR = ^M;
LF = ^J;
{
 Type Declarations
}
type Symbol = string[8];
SymTab = array[1..1000] of Symbol;
TabPtr = ^SymTab;
{
 Variable Declarations
}
var Look : char;  Lookahead Character
Lcount: integer;  Label Counter
{
 Read New Character From Input Stream
}
procedure GetChar;
begin
Read(Look);
end;
{
 Report an Error
}
procedure Error(s: string);
begin
WriteLn;
WriteLn(^G, 'Error: ', s, '.');
end;
{
 Report Error and Halt
}
procedure Abort(s: string);
begin
Error(s);
Halt;
end;
{
 Report What Was Expected
}
procedure Expected(s: string);
begin
Abort(s + ' Expected');
end;
{
 Recognize an Alpha Character
}
function IsAlpha(c: char): boolean;
begin
IsAlpha := UpCase(c) in ['A'..'Z'];
end;
{
 Recognize a Decimal Digit
}
function IsDigit(c: char): boolean;
begin
IsDigit := c in ['0'..'9'];
end;
{
 Recognize an AlphaNumeric Character
}
function IsAlNum(c: char): boolean;
begin
```

```
IsAlNum := IsAlpha(c) or IsDigit(c);
end;
–
 Recognize an Addop
function IsAddop(c: char): boolean;
begin
IsAddop := c in ['+', '-'];
end;
–
 Recognize a Mulop
function IsMulop(c: char): boolean;
begin
IsMulop := c in ['*', '/'];
end;
–
 Recognize White Space
function IsWhite(c: char): boolean;
begin
IsWhite := c in [' ', TAB];
end;
–
 Skip Over Leading White Space
procedure SkipWhite;
begin
while IsWhite(Look) do
GetChar;
end;
–
 Match a Specific Input Character
procedure Match(x: char);
begin
if Look &lt;&gt; x then Expected("" + x + "");
GetChar;
SkipWhite;
end;
–
 Skip a CRLF
procedure Fin;
begin
if Look = CR then GetChar;
if Look = LF then GetChar;
SkipWhite;
end;
–
 Get an Identifier
function GetName: char;
begin
while Look = CR do
Fin;
if not IsAlpha(Look) then Expected('Name');
Getname := UpCase(Look);
GetChar;
SkipWhite;
end;
–
 Get a Number
function GetNum: char;
```

```
begin
if not IsDigit(Look) then Expected('Integer');
GetNum := Look;
GetChar;
SkipWhite;
end;
{--------------------------------------------------------------}
{ Generate a Unique Label }
function NewLabel: string;
var S: string;
begin
Str(LCount, S);
NewLabel := 'L' + S;
Inc(LCount);
end;
{--------------------------------------------------------------}
{ Post a Label To Output }
procedure PostLabel(L: string);
begin
WriteLn(L, ':');
end;
{--------------------------------------------------------------}
{ Output a String with Tab }
procedure Emit(s: string);
begin
Write(TAB, s);
end;
{--------------------------------------------------------------}
{ Output a String with Tab and CRLF }
procedure EmitLn(s: string);
begin
Emit(s);
WriteLn;
end;
{--------------------------------------------------------------}
{ Parse and Translate an Identifier }
procedure Ident;
var Name: char;
begin
Name := GetName;
if Look = '(' then begin
Match('(');
Match(')');
EmitLn('BSR ' + Name);
end
else
EmitLn('MOVE ' + Name + '(PC),D0');
end;
{--------------------------------------------------------------}
{ Parse and Translate a Math Factor }
procedure Expression; Forward;
procedure Factor;
begin
if Look = '(' then begin
Match('(');
Expression;
Match(')');
```

```
end
else if IsAlpha(Look) then
Ident
else
EmitLn('MOVE #' + GetNum + ',D0');
end;
_
 Parse and Translate the First Math Factor
procedure SignedFactor;
var s: boolean;
begin
s := Look = '-';
if IsAddop(Look) then begin
GetChar;
SkipWhite;
end;
Factor;
if s then
EmitLn('NEG D0');
end;
_
 Recognize and Translate a Multiply
procedure Multiply;
begin
Match('*');
Factor;
EmitLn('MULS (SP)+,D0');
end;
_
 Recognize and Translate a Divide
procedure Divide;
begin
Match('/');
Factor;
EmitLn('MOVE (SP)+,D1');
EmitLn('EXS.L D0');
EmitLn('DIVS D1,D0');
end;
_
 Completion of Term Processing (called by Term and FirstTerm
procedure Term1;
begin
while IsMulop(Look) do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'*': Multiply;
'/': Divide;
end;
end;
end;
_
 Parse and Translate a Math Term
procedure Term;
begin
Factor;
Term1;
end;
```

```
_
 Parse and Translate a Math Term with Possible Leading Sign
procedure FirstTerm;
begin
SignedFactor;
Term1;
end;
_
 Recognize and Translate an Add
procedure Add;
begin
Match('+');
Term;
EmitLn('ADD (SP)+,D0');
end;
_
 Recognize and Translate a Subtract
procedure Subtract;
begin
Match('-');
Term;
EmitLn('SUB (SP)+,D0');
EmitLn('NEG D0');
end;
_
 Parse and Translate an Expression
procedure Expression;
begin
FirstTerm;
while IsAddop(Look) do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'+': Add;
'-': Subtract;
end;
end;
end;
_
 Parse and Translate a Boolean Condition
 This version is a dummy
Procedure Condition;
begin
EmitLn('Condition');
end;
_
 Recognize and Translate an IF Construct
procedure Block;
Forward;
procedure DoIf;
var L1, L2: string;
begin
Match('i');
Condition;
L1 := NewLabel;
L2 := L1;
EmitLn('BEQ ' + L1);
Block;
```

```
if Look = 'l' then begin
Match('l');
L2 := NewLabel;
EmitLn('BRA ' + L2);
PostLabel(L1);
Block;
end;
PostLabel(L2);
Match('e');
end;
_
 Parse and Translate an Assignment Statement
procedure Assignment;
var Name: char;
begin
Name := GetName;
Match('=');
Expression;
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE D0,(A0)');
end;
_
 Recognize and Translate a Statement Block
procedure Block;
begin
while not(Look in ['e', 'l']) do begin
case Look of
'i': DoIf;
CR: while Look = CR do
Fin;
else Assignment;
end;
end;
end;
_
 Parse and Translate a Program
procedure DoProgram;
begin
Block;
if Look &lt;&gt; 'e' then Expected('END');
EmitLn('END')
end;
_
 Initialize
procedure Init;
begin
LCount := 0;
GetChar;
end;
_
 Main Program
begin
Init;
DoProgram;
end.
_
```

Пара комментариев:

Форма синтаксического анализатора выражений, использующего FirstTerm и т.п., немного отличается от того, что вы видели ранее. Это еще одна вариация на ту же самую тему. Не позволяйте им вертеть вами... изменения необязательны для того, что будет дальше.

Заметьте, что как обычно я добавил вызовы Fin в стратегических местах для поддержки множественных строк.

Прежде чем приступить к добавлению сканера, сначала скопируйте этот файл и проверьте, что он действительно корректно выполняет анализ. Не забудьте «кода»: "i" для IF, "l" для ELSE и "e" для ELSE или ENDIF.

Если программа работает, тогда давайте поспешим. При добавлении модулей сканера в программу поможет систематический план. Во всех синтаксических анализаторах, которые мы написали до этого времени, мы придерживались соглашения, что текущий предсказывающий символ должен всегда быть непустым символом. Мы предварительно загружали предсказывающий символ в Init и после этого оставляли «помпу запущенной». Чтобы позволить программе работать правильно с новыми строками мы должны ее немного модифицировать и обрабатывать символ новой строки как допустимый токен.

В многосимвольной версии правило аналогично: текущий предсказыващий символ должен всегда оставаться на начале следующей лексемы или на новой строке.

Многосимвольная версия показана ниже. Чтобы получить ее я сделал следующие изменения:

• Добавлены переменные Token и Value и определения типов, необходимые для Lookup.

• Добавлено определение KWList и KWcode.

• Добавлен Lookup.

• GetName и GetNum заменены их многосимвольными версиями. (Обратите внимание, что вызов Lookup был перемещен из GetName, так что он не будет выполняться внутри выражений).

• Создана новая, рудиментарная Scan, которая вызывает GetName затем сканирует ключевые слова.

• Создана новая процедура MatchString, которая ищет конкретное ключевое слово. Заметьте, что в отличие от Match, MatchString не считывает следующее ключевое слово.

• Изменен Block для вызова Scan.

• Немного изменены вызовы Fin. Fin теперь вызывается из GetName.

Программа полностью:

```
{--------------------------------------------------------------}
program KISS;
{--------------------------------------------------------------}
{ Constant Declarations }
const TAB = ^I;
CR = ^M;
LF = ^J;
{--------------------------------------------------------------}
{ Type Declarations }
type Symbol = string[8];
SymTab = array[1..1000] of Symbol;
TabPtr = ^SymTab;
{--------------------------------------------------------------}
{ Variable Declarations }
var Look : char;  { Lookahead Character }
Token : char;  { Encoded Token }
Value : string[16];  { Unencoded Token }
Lcount: integer;  { Label Counter }
{--------------------------------------------------------------}
{ Definition of Keywords and Token Types }
```

```
const KWlist: array [1..4] of Symbol =
('IF', 'ELSE', 'ENDIF', 'END');
const KWcode: string[5] = 'xilee';
–
 Read New Character From Input Stream
procedure GetChar;
begin
Read(Look);
end;
–
 Report an Error
procedure Error(s: string);
begin
WriteLn;
WriteLn(^G, 'Error: ', s, '.');
end;
–
 Report Error and Halt
procedure Abort(s: string);
begin
Error(s);
Halt;
end;
–
 Report What Was Expected
procedure Expected(s: string);
begin
Abort(s + ' Expected');
end;
–
 Recognize an Alpha Character
function IsAlpha(c: char): boolean;
begin
IsAlpha := UpCase(c) in ['A'..'Z'];
end;
–
 Recognize a Decimal Digit
function IsDigit(c: char): boolean;
begin
IsDigit := c in ['0'..'9'];
end;
–
 Recognize an AlphaNumeric Character
function IsAlNum(c: char): boolean;
begin
IsAlNum := IsAlpha(c) or IsDigit(c);
end;
–
 Recognize an Addop
function IsAddop(c: char): boolean;
begin
IsAddop := c in ['+', '-'];
end;
–
 Recognize a Mulop
function IsMulop(c: char): boolean;
begin
```

```
IsMulop := c in ['*', '/'];
end;
–
 Recognize White Space
function IsWhite(c: char): boolean;
begin
IsWhite := c in [' ', TAB];
end;
–
 Skip Over Leading White Space
procedure SkipWhite;
begin
while IsWhite(Look) do
GetChar;
end;
–
 Match a Specific Input Character
procedure Match(x: char);
begin
if Look &lt;&gt; x then Expected("" + x + "");
GetChar;
SkipWhite;
end;
–
 Skip a CRLF
procedure Fin;
begin
if Look = CR then GetChar;
if Look = LF then GetChar;
SkipWhite;
end;
–
 Table Lookup
function Lookup(T: TabPtr; s: string; n: integer): integer;
var i: integer;
found: boolean;
begin
found := false;
i := n;
while (i &gt; 0) and not found do
if s = T^[i] then
found := true
else
dec(i);
Lookup := i;
end;
–
 Get an Identifier
procedure GetName;
begin
while Look = CR do
Fin;
if not IsAlpha(Look) then Expected('Name');
Value := '';
while IsAlNum(Look) do begin
Value := Value + UpCase(Look);
GetChar;
```

```
end;
SkipWhite;
end;
–
 Get a Number
procedure GetNum;
begin
if not IsDigit(Look) then Expected('Integer');
Value := '';
while IsDigit(Look) do begin
Value := Value + Look;
GetChar;
end;
Token := '#';
SkipWhite;
end;
–
 Get an Identifier and Scan it for Keywords
procedure Scan;
begin
GetName;
Token := KWcode[Lookup(Addr(KWlist), Value, 4) + 1];
end;
–
 Match a Specific Input String
procedure MatchString(x: string);
begin
if Value &lt;&gt; x then Expected('' + x + '');
end;
–
 Generate a Unique Label
function NewLabel: string;
var S: string;
begin
Str(LCount, S);
NewLabel := 'L' + S;
Inc(LCount);
end;
–
 Post a Label To Output
procedure PostLabel(L: string);
begin
WriteLn(L, ':');
end;
–
 Output a String with Tab
procedure Emit(s: string);
begin
Write(TAB, s);
end;
–
 Output a String with Tab and CRLF
procedure EmitLn(s: string);
begin
Emit(s);
WriteLn;
end;
```

```
_
 Parse and Translate an Identifier
procedure Ident;
begin
GetName;
if Look = '(' then begin
Match('(');
Match(')');
EmitLn('BSR ' + Value);
end
else
EmitLn('MOVE ' + Value + '(PC),D0');
end;
_
 Parse and Translate a Math Factor
procedure Expression; Forward;
procedure Factor;
begin
if Look = '(' then begin
Match('(');
Expression;
Match(')');
end
else if IsAlpha(Look) then
Ident
else begin
GetNum;
EmitLn('MOVE #' + Value + ',D0');
end;
end;
_
 Parse and Translate the First Math Factor
procedure SignedFactor;
var s: boolean;
begin
s := Look = '-';
if IsAddop(Look) then begin
GetChar;
SkipWhite;
end;
Factor;
if s then
EmitLn('NEG D0');
end;
_
 Recognize and Translate a Multiply
procedure Multiply;
begin
Match('*');
Factor;
EmitLn('MULS (SP)+,D0');
end;
_
 Recognize and Translate a Divide
procedure Divide;
begin
Match('/');
```

```
Factor;
EmitLn('MOVE (SP)+,D1');
EmitLn('EXS.L D0');
EmitLn('DIVS D1,D0');
end;
_
 Completion of Term Processing (called by Term and FirstTerm
procedure Term1;
begin
while IsMulop(Look) do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'*': Multiply;
'/': Divide;
end;
end;
end;
_
 Parse and Translate a Math Term
procedure Term;
begin
Factor;
Term1;
end;
_
 Parse and Translate a Math Term with Possible Leading Sign
procedure FirstTerm;
begin
SignedFactor;
Term1;
end;
_
 Recognize and Translate an Add
procedure Add;
begin
Match('+');
Term;
EmitLn('ADD (SP)+,D0');
end;
_
 Recognize and Translate a Subtract
procedure Subtract;
begin
Match('-');
Term;
EmitLn('SUB (SP)+,D0');
EmitLn('NEG D0');
end;
_
 Parse and Translate an Expression
procedure Expression;
begin
FirstTerm;
while IsAddop(Look) do begin
EmitLn('MOVE D0,-(SP)');
case Look of
'+': Add;
```

```
'-': Subtract;
end;
end;
end;
_
 Parse and Translate a Boolean Condition
 This version is a dummy
Procedure Condition;
begin
EmitLn('Condition');
end;
_
 Recognize and Translate an IF Construct
procedure Block; Forward;
procedure DoIf;
var L1, L2: string;
begin
Condition;
L1 := NewLabel;
L2 := L1;
EmitLn('BEQ ' + L1);
Block;
if Token = 'l' then begin
L2 := NewLabel;
EmitLn('BRA ' + L2);
PostLabel(L1);
Block;
end;
PostLabel(L2);
MatchString('ENDIF');
end;
_
 Parse and Translate an Assignment Statement
procedure Assignment;
var Name: string;
begin
Name := Value;
Match('=');
Expression;
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE D0,(A0)');
end;
_
 Recognize and Translate a Statement Block
procedure Block;
begin
Scan;
while not (Token in ['e', 'l']) do begin
case Token of
'i': DoIf;
else Assignment;
end;
Scan;
end;
end;
_
 Parse and Translate a Program
```

```
procedure DoProgram;
begin
Block;
MatchString('END');
EmitLn('END')
end;
–
 Initialize
procedure Init;
begin
LCount := 0;
GetChar;
end;
–
 Main Program
begin
Init;
DoProgram;
end.
–
```

Сравните эту программу с ее односимвольным вариантом. Я думаю вы согласитесь, что различия минимальны.

ЗАКЛЮЧЕНИЕ

К этому времени вы узнали как анализировать и генерировать код для выражений, булевых выражений и управляющих структур. Теперь вы изучили, как разрабатывать лексические анализаторы и как встроить их элементы в транслятор. Вы все еще не видели всех элементов, объединеных в одну программу, но на основе того, что мы сделали ранее вы должны прийти к заключению, что легко расширить наши ранние программы для включения лексических анализаторов.

Мы очень близки к получению всех элементов, необходимых для построения настоящего, функционального компилятора. Есть еще несколько отсутствующих вещей, особенно вызовы процедур и определения типов. Мы будем работать с ними на следующих нескольких уроках. Прежде чем сделать это, однако, я подумал что было бы забавно превратить транслятор в настоящий компилятор. Это то, чем мы займемся в следующей главе.

До настоящего времени мы применяли предпочтительно восходящий метод синтаксического анализа, начиная с низкоуровневых конструкций и продвигаясь вверх. В следующей главе я также взгляну сверху вниз, и мы обсудим, как изменяется структура транслятора при изменении определения языка.

Увидимся.

Немного философии

ВВЕДЕНИЕ

Этот урок будет отличаться от других уроков в нашей серии по синтаксическому анализу и конструированию компиляторов. На этом уроке не будет никаких экспериментов или кода. На этот раз я хотел бы просто поговорить с вами некоторое время. К счастью, это будет короткий урок и затем мы сможем продолжить с того места где остановились, надо надеяться с обновленной энергией.

Когда я учился в университете, я обнаружил, что могу всегда следить за профессорской лекцией намного лучше, если знал куда он идет. Готов поспорить, с вами было то же самое.

Так что я подумал, может быть пришло время расказать вам куда мы идем с этой серией: что нас ждет в будущих главах и вообще что к чему. Я также поделюсь своими общими мыслями о полезности того, что мы делали.

ДОРОГА ДОМОЙ

Пока что мы охватили синтаксический анализ и трансляцию арифметических выражений, булевых выражений и их комбинаций, связанных операторами отношений. Мы также сделали то же самое для управляющих конструкций. Во всем этом мы склонялись в основном к использованию нисходящего синтаксического анализа методом рекурсивного спуска, определение синтаксиса в БНФ и непосредственной генерации ассемблерного кода. Мы также изучили значение такого приема как односимвольные токены. В последней главе мы работали с лексическим анализом и я показал вам простой но мощный способ преодоления односимвольного барьера.

В течение всех этих исследований, я особенно выделял философию KISS... Keep It Simple, Sidney... и я надеюсь, что к настоящему времени вы поняли, насколько простыми могут в действительности быть эти вещи. Хотя наверняка имеются области в теории компиляции которые являются по настоящему пугающими, основной мыслью этой серии является то, что на практике вы можете просто вежливо обойти многие из этих областей. Если определение языка способствует этому или, как в этой серии, если вы можете определить язык по ходу дела, то возможно записать определение языка в БНФ с достаточным удобством. И, как мы видели, вы можете вывести процедуры синтаксического анализа из БНФ почти также быстро, как вы можете набирать на клавиатуре.

По мере того, как наш компилятор принимал некоторую форму, он приобретал больше частей, но каждая часть довольно мала и проста и очень похожа на все другие.

К этому моменту у нас есть многое из того, что составляет настоящий практический компилятор. Фактически, мы уже имеем все что нам нужно для создания игрушечного языка столь же мощного, как, скажем, Tiny Basic. В следующих двух главах мы пойдем вперед и определим этот язык.

Для завершения этой серии, у нас все еще есть несколько тем для раскрытия. Они включают:

- Вызовы процедур, с параметрами и без.
- Локальные и глобальные переменные.
- Базовые типы, такие как символьные и целочисленные типы.
- Массивы.
- Строки.
- Типы и структуры, определяемые пользователем.
- Синтаксические анализаторы с деревьями и промежуточные языки.
- Оптимизация.

Все это будет рассмотрено в будущих главах. Когда мы закончим, вы будете иметь все инструменты, необходимые для разработки и создания своего собственного языка и компиляторов для его трансляции.

Я не могу спроектировать эти языки для вас, но я могу дать некоторые комментарии и рекомендации. Я уже высказал некоторые из них в прошлых главах. Вы видели, например, какие управляющие структуры я предпочитаю.

Эти конструкции будут частью создаваемых мной языков. К этому моменту я представляю три языка, два из которых вы увидите в очередных главах:

TINY – минимальный, но пригодный для использования язык уровня Tiny Basic или

Tiny C. Он не будет очень практичным, но будет достаточно мощным, чтобы позволить вам писать и запускать настоящие программы которые делают что-нибудь заслуживающее внимание.

KISS – язык, который я создаю для своего собственного использования. KISS предназначен быть языком системного программирования. Он не будет иметь строгого контроля типов или причудливых структур данных, но он будет поддерживать большинство вещей, которые я хочу делать с языком более высокого уровня (HOL), за исключением возможно написания компиляторов.

Я также играл в течение нескольких лет с идеей HOL-подобного ассемблера со структурными управляющими конструкциями и HOL-подобными операциями присваивания. Это фактически было стимулом для моего первоначального углубления в джунгли теории компиляции. Этот язык возможно никогда не будет создан просто потому, что я узнал, что проще реализовать язык типа KISS, который использует только подмножество инструкций ЦПУ. Как вы знаете, ассемблер может быть предельно причудливым и нерегулярным, и язык, который отображается в него один к одному, может быть настоящим вызовом. Однако я всегда чувствовал, что синтаксис, используемый в стандартных ассемблерах тупой... почему

MOVE.L A,B

лучше или проще для трансляции, чем

B=A?

Я думаю, было бы интересным упражнением разработка «компилятора» который дал бы программисту полный доступ и к контролю над полным набором инструкций ЦПУ, и позволил бы вам генерировать программы настолько же эффективные как язык ассемблер без болезненного изучения набора мнемоник. Это может быть сделано? Я не знаю. Настоящим вопросом может быть вопрос «будет ли полученный язык проще, чем ассемблер?» Если нет, то в нем нет никакого смысла. Я думаю, что это может быть сделано, но я полностью еще не уверен в том, как должен выглядеть синтаксис.

Возможно у вас есть некоторые комментарии или предложения об этом. Буду рад услышать их.

Вы возможно не будете удивлены узнав, что уже работал в большинстве тех областей, которые мы рассмотрим. Я имею несколько хороших новостей: дела никогда не будут намного более сложными, чем они были до этого. Возможно построить завершенный, работающий компилятор для реального языка используя только те самые методы которые вы изучили до этого. И это поднимет некоторые интересные вопросы.

ПОЧЕМУ ЭТО ТАК ПРОСТО?

Перед осуществлением этой серии я всегда думал, что компиляторы были просто естественно сложными компьютерными программами... предельно вызывающими. Однако то, что мы здесь делали обычно оказывалось совершенно простым, иногда даже тривиальным.

Некоторое время я думал, что это было просто потому, что я еще не залез в глубь темы. Я только охватил простые части. Я легко признаюсь вам что даже когда я начинал эту серию я не был уверен в том, как далеко мы будем способны продвинуться прежде чем дела станут слишком сложными для работы имеющимися способами. Но сейчас я уже нахожусь достаточно близко, чтобы увидеть конец пути. Какой вывод?

ЗДЕСЬ НЕТ НИЧЕГО СЛОЖНОГО!

Затем я думал, что причина в том, что мы не генерировали очень хороший объектный код. Те из вас, кто следовали этой серии и пытались компилировать примеры, знают, что хотя код работает и достаточно отказоустойчив, его эффективность довольно ужасна. Я

подчеркивал, что если бы мы сконцентрировались на получении компактного кода, то быстро бы получили всю недостающую сложность.

В какой то степени это так. В частности, мои первые небольшие усилия при попытке повысить эффективность подняли сложность до опасного уровня. Но с той поры когда я возился с некоторыми простыми методами оптимизацией и обнаружил некоторые, которые приводят к очень приличному качеству кода без добавления больших сложностей.

Наконец я подумал, что возможно причина была в «игрушечной» природе компилятора Я не претендовал на то, что мы когда-нибудь будем способны построить компилятор, конкурирующий с Borland и Microsoft. И однако снова, когда я забираюсь глубже в эти дела различия начинают стираться.

Просто чтобы удостовериться что до вас дошла эта мысль, позвольте мне ее высказать напрямую:

Используя методы, которые мы здесь применяли, возможно создать работающий, промышленного качества компилятор не добавляя много сложности к тому, что мы уже сделали.

С тех пор, как началась эта серия, я получил от вас некоторые комментарии. Большинство из них повторяют мои собственные мысли: «Это просто! Почему учебники представляют это настолько сложным?» Хороший вопрос.

Недавно я возвратился и взглянул на некоторые из этих текстов снова и даже купил и читаю некоторые новые. Каждый раз я возвращался с тем же чувством: эти ребята представляют это слишком сложным.

Что происходит? Почему все это кажется сложным в этих книгах, но легким для нас? Действительно ли мы умней чем Ахо, Ульман, Бринч Хансен и все остальные?

Едва ли. Но мы делаем некоторые вещи по-другому и все более и более я начинаю ценить значение нашего подхода и способ, которым он упрощает дело. Кроме очевидных сокращений, которые я выделил в первой части, типа односимвольных токенов и консольного ввода/вывода, мы сделали некоторые неявные предположения и сделали некоторые вещи отличными от того, как разрабатывали компиляторы в прошлом. Как оказалось, наш метод делает жизнь намного проще.

Но почемы все другие ребята не используют его?

Вы должны вспомнить контекст некоторых ранних разработок компиляторов. Эти люди работали на очень небольших компьютерах с ограниченными возможностями. Объемы памяти были очень ограничены, набор команд центрального процессора был минимален и программы чаще выполнялись в пакетном режиме, чем в интерактивном. Как оказалось, это повлияло на некоторые ключевые решения проекта, которые действительно усложнили проект. До недавнего времени я не понимал, насколько классический дизайн компилятора был обусловлен доступным оборудованием.

Даже в тех случаях, где эти ограничения больше не накладывались, люди предпочитали структурировать их программы тем же самым образом, так как это способ, которому они обучались.

В нашем случае мы начали с чистого листа бумаги. Имеется опасность, конечно, что вы попадетесь в ловушки, которые другие люди давно научились избегать. Но это также позволило нам использовать различные подходы, которые, частично из-за проекта, частично из-за чистой удачи, позволили нам добиться простоты.

Имеются области, которые, я думаю, в прошлом приводили к сложности:

Ограниченная оперативная память, вынуждающая выполнять множество проходов.

Я только что прочитал «Brinch Hansen on Pascal Compilers» (отличная книга, BTW). Он разработал компилятор Pascal для PC, но он начал в 1981 г. с систем с 64K памяти и поэтому почти каждое решение проекта который он делал, было нацелено на то, чтобы уместить компилятор в ОЗУ. Чтобы сделать это, его компилятор выполнял три прохода, один из которых – лексический анализ. Не было никакого способа, с помощью которого он мог бы,

например, использовать распределенный сканер, который я представил в последней главе, потому что структура программы не позволяла этого. Ему также требовались не один а два промежуточных языка для обеспечения связи между фазами.

Все ранние создатели компиляторов были вынуждены иметь дело с такой проблемой: разбить компилятор на достаточные части так, чтобы они поместились в памяти. Когда у вас есть множество проходов, вы должны добавить структуры данных для поддержки информации которую каждый проход оставляет для следующего. Это добавляет сложность и завершает управление проектом. В книге Ли «The Anatomy of a Compiler» упоминается компилятор Fortran, разработанный для IBM 1401. Он имел не менее 63 отдельных проходов! Само собой разумеется, в компиляторе, подобном этому, разделение на фазы доминировало бы над дизайном.

Даже в ситуации, когда ОЗУ достаточно, люди предпочитали использовать те же самые методы, с которыми они хорошо знакомы. До тех пор, пока не появился Turbo Pascal, мы не знали насколько может быть простым компилятор если бы вы начали с других предположений.

Пакетная обработка.

В ранние дни пакетная обработка была единственным выбором... не существовало никаких интерактивных вычислений. Даже сегодня компиляторы по существу выполняются в пакетном режиме.

В компиляторах для майнфреймов, так же как и во многих микро компиляторах, значительные усилия расходуются на восстановление после ошибок... это может занять 30-40% компилятора и полностью управлять проектом. Идея состоит в том, чтобы избежать остановки на первой ошибке, а скорее идти любой ценой, так чтобы вы могли сказать программисту о как можно большем количестве ошибок во всей программе насколько возможно.

Все это возвращает нас к дням ранних майнфреймов, где время выполнения измерялось в часах и днях и было важно выжать каждую последнюю унцию информации из каждого выполнения.

В этой серии я был очень осторожен и избежал проблемы восстановления после ошибок и вместо этого наш компилятор просто останавливается с сообщением на первой ошибке. Я откровенно признаюсь, что это было в основном потому, что я захотел использовать легкий путь и сохранить простоту. Но этот метод, заданный изначально Borland в Turbo Pascal также имеет много полезного в любом случае. Кроме сохранения простоты компилятора это также очень хорошо соответствует идее интерактивной системы. Когда компиляция происходит быстро и, особенно, когда вы имеете редактор типа Borland который будет правильно указывать вам на точку ошибки, тогда имеет смысл остановиться там и просто перезапустить компиляцию после того, как ошибка исправлена.

Большие программы.

Ранние компиляторы были разработаны для поддержки больших программ... по существу бесконечных. В те дни существовал небольшой выбор; идея с библиотеками подпрограмм и раздельной компиляцией была еще в будущем. Снова, это предположение вело к многопроходному дизайну и промежуточным файлам для поддержки результатов раздельной обработки.

Поставленная Бринч Хансеном цель состояла в том, чтобы компилятор был способен компилировать сам себя. Снова, из-за ограничений оперативной памяти это приводило его к многопроходному дизайну. Он нуждался в таком маленьком резидентном коде компилятора, насколько возможно, так чтобы необходимые таблицы и другие структуры данных поместились в оперативную память.

Я не заявил об этом пока, потому что не было необходимости... мы всегда просто читали и записывали данные как потоки, в любом случае. Но, для заметки, мой план всегда был в том, чтобы в промышленном компиляторе исходные и обьектные данные должны сосуществовать в ОЗУ с компилятором, аля ранний Turbo Pascal. Вот почему я был

осторожен и сохранил подпрограммы типа GetChar и Emit как отдельные попрограммы, несмотря на их небольшой размер. Будет просто изменить их на чтение и запись из памяти.

Акцент на эффективность.

Джон Бэкус заявил, что когда он и его коллеги разработали первоначальный компилятор Fortran они знали, что они должны получать компактный код. В те дни имелись сильные чувства против HOL в пользу ассемблера и причиной была эффективность. Если бы Fortran не производил очень хороший код по стандартам ассемблера, пользователи просто бы отказались использовать его. Заметьте, компилятор Fortran оказался одним из наиболее эффективных из когда либо созданных.в терминах качества кода. Но он был сложным!

Сегодня мы имеем мощъ ЦПУ и размер ОЗУ с запасом, так что эффективность кода не такая большая проблема. Старательно игнорируя эту проблему мы действительно были способны сохранить простоту. Как ни странно, тем не менее, как я сказал, я нашел некоторую оптимизацию которую мы можем добавить в базовую структуру компилятора не добавляя слишком много сложности. Так что в этом случае мы получим свой пирог и сьедим его: мы в любом случае закончим с приемлемым качеством кода.

Ограниченный набор инструкций.

Первые компьютеры имели примитивный набор команд. Вещи, которые мы считаем само собой разумеющимися такие как операции со стеком и косвенная адресация появились с большими сложностями.

Пример: в большинстве компиляторов имеется структура данных, называемая литерный пул (literal pool). Компилятор обычно идентифицирует все литералы, используемые в программе и собирает их в одиночную структуру данных. Все ссылки на литералы сделаны косвенно на этот пул. В конце компиляции компилятор выдает команды для выделения памяти и инициализации литерного пула.

Нам пока не нужно было обращаться к этой проблеме. Когда нам нужно загрузить литерал мы просто делаем это строкой:

```
MOVE #3,D0
```

Можно кое-что упомянуть об использования литерного пула особенно на машине типа 8086, где данные и код могут быть разделены. Однако все это добавляет довольно большое количество сложности с небольшим результатом.

Конечно, без стека мы бы потерялись. И вызовы подпрограмм и временная память сильно зависят от стека и мы использовали его даже больше, чем необходимо для облегчения синтаксического анализа выражений.

Желание общности.

Многое из содержимого типичной книги по компиляторам акцентировано на вопросы, к которым мы совсем не обращались... вопросы типа автоматической трансляции грамматик или генерация таблиц LALR анализа. Это не просто, потому что авторы хотят впечатлить вас. Имеются хорошие практические причины, почему эти темы рассмотрены здесь.

Мы концентрировались на использовании синтаксического анализатора с рекурсивным спуском для анализа детерминированной грамматики, т.е. грамматики, которая однозначна и, следовательно, может быть проанализирована с одним уровнем предсказывания. Я не сделал из этого большого ограничения, но факт то, что это представляет небольшое подмножество возможных грамматик. Фактически, существует бесконечное число грамматик, которые мы не можем анализировать используя наш метод. LR метод более мощный и может работать с теми грамматиками, с которыми мы не можем.

В теории компиляции важно знать, как работать с этими другими грамматиками и как преобразовать их в грамматики которые проще для работы с ними. К примеру многие (но не все) неоднозначные грамматики могут быть преобразованы в однозначные. Способ сделать это не всегда очевиден, всетаки, и так много людей посвятили годы на разработку способа их автоматического преобразования.

На практике, эти проблемы оказываются значительно менее важными. Современные языкы стараются разрабатывать так, чтобы они были простыми для анализа в любом случае.

Это было ключевой мотивацией при разработке Pascal. Несомненно, имеются паталогические грамматики, для которых вы с большим трудом написали бы однозначную БНФ, но в реальном мире лучшим ответом возможно было бы избежание этих грамматик.

В нашем случае, конечно, мы трусливо позволили языку развиваться по ходу дела. Вы не можете всегда иметь такую роскошь. Однако, с небольшой заботой вы были бы способны сохранить синтаксический анализатор простым без необходимости прибегать к автоматическому переводу грамматик.

В этой серии мы приняли значительно отличающийся подход. Мы начали с чистого листа бумаги и разработали методы, которые работают в том контексте, в котором мы находимся: это однопользовательский персональный компьютер с вполне достаточно мощным ЦПУ и объемом ОЗУ. Мы ограничили сами себя приемлемыми грамматиками, которые легки для анализа, мы с успехом использовали систему команд ЦПУ, и мы не концентрировались на эффективности. Именно поэтому это было просто.

Означает ли это, что мы навсегда обречены создавать только игрушечные компиляторы? Нет, я так не думаю. Я уже сказал, что мы можем добавить некоторую оптимизацию без изменения структуры компилятора. Если мы захотим обрабатывать большие файлы, мы всегда можем добавить для этого буферизацию файлов. Эти вещи не оказывают влияния на общий дизайн компилятора.

И я думаю что это главный фактор. Начав с маленьких и ограниченных случаев мы были способны сконцентрироваться на структуре компилятора, которая естественна для работы. Так как структура естественным образом удовлетворяет работе, она почти обречена быть простой и прозрачной. Добавление возможностей не должно изменять основную структуру. Мы можем просто добавить расширения типа файловой структуры или добавить уровень оптимизации. Я считаю, что когда ресурсы были ограничены, структуры, которые люди получали, были искуственно искажены чтобы заставить их работать в этих условиях, и не были оптимальными структурами для имеющейся проблемы.

ЗАКЛЮЧЕНИЕ

В любом случае, это мое личное предположение каким образом у нас была возможность сохранить простоту. Мы начали с чего-то простого и позволили ему развиваться естественным образом, не пытаясь направит его в какое-то традиционное русло.

Мы собираемся продолжать таким же образом. Я дал вам список областей, которые мы охватим в следующих главах. После прочтения этих глав вы будете способны создавать законченные, работающие компиляторы почти для любого случая и делать это легко. Если вы действительно хотите создать компилятор промышленного качества вы сможете сделать и это также.

Для тех из вас, кто застоялся в ожидании кода для синтаксического анализатора, я приношу извинения за это отклонение. Я просто подумал, что вы хотели бы немного рассмотреть дела в перспективе. В следующий раз мы вернемся к основной цели обучения.

Пока что мы рассмотрели только части компиляторов и хотя мы имеем многое из завершенного языка мы не говорили о том как сложить все это вместе. Это будет темой наших следующих двух глав. Затем мы поспешим к новым темам, которые я указал в начале этой главы.

Увидимся.

Вид сверху

ВВЕДЕНИЕ

В предыдущих главах мы изучили многие из методов, необходимых для создания

полноценного компилятора. Мы разработали операции присваивания (с булевыми и арифметическими выражениями), операторы отношений и управляющие конструкции. Мы все еще не обращались к вопросу вызова процедур и функций, но даже без них мы могли бы в принципе создать мини-язык. Я всегда думал, что было бы забавно просто посмотреть, насколько маленьким можно было бы построить язык, чтобы он все еще оставался полезным. Теперь мы уже почти готовы сделать это. Существует проблема: хотя мы знаем, как анализировать и транслировать конструкции, мы все еще совершенно не знаем, как сложить их все вместе в язык.

В этих ранних главах разработка наших программ имела явно восходящий характер. В случае с синтаксическим анализом выражений, например, мы начали с самых низкоуровневых конструкций, индивидуальных констант и переменных и прошли свой путь до более сложных выражений.

Большинство людей считают, что нисходящий способ разработки лучше, чем восходящий. Я тоже так думаю, но способ, который мы использовали, казался естественно достаточным для тех вещей, которые мы анализировали.

Тем не менее вы не должны думать, что последовательный подход, который мы применяли во всех этих главах, является принципиально восходящим. В этой главе я хотел бы показать вам, что этот подход может работать точно также, когда применяется сверху вниз... может быть даже лучше. Мы рассмотрим языки типа C и Pascal и увидим как могут быть построены законченные компиляторы начиная сверху.

В следующей главе мы применим ту же самую методику для создания законченного транслятора подмножества языка KISS, который я буду называть TINY. Но одна из моих целей в этой серии состоит в том, чтобы вы не только могли увидеть как работает компилятор для TINY или KISS, но чтобы вы также могли разрабатывать и создавать компиляторы своих собственных языков. Примеры Си и Паскаля помогут вам в этом. Одна вещь, которую я хотел чтобы вы увидели, состоит в том, что естественная структура компилятора очень сильно зависит от транслируемого языка, поэтому простота и легкость конструирования компилятора очень сильно зависит от того, позволите ли вы языку определять структуру программы.

Немного сложнее получить полный компилятор C или Pascal, да мы и не будем. Но мы можем расчистить верхние уровни так, чтобы вы увидели как это делается.

Давайте начнем.

ВЕРХНИЙ УРОВЕНЬ

Одна из самых больших ошибок людей при нисходящем проектировании заключается в неправильном выборе истинной вершины. Они думают, что знают какой должна быть общая структура проекта и поэтому они продолжают и записывают ее.

Всякий раз, когда я начинаю новый проект, я всегда хочу сделать это в самом начале. На языке разработки программ (program design language – PDL) этот верхний уровень походит на что-нибудь вроде:

```
begin
solve the problem
end
```

Конечно, я соглашусь с вами, что это не слишком большая подсказка о том, что расположено на следующем уровене, но я все равно запишу это просто для того, чтобы почувствовать, что я действительно начинаю с вершины.

В нашем случае, общая функция компилятора заключается в компиляции законченной программы. С этого начинается любое определение языка, записанное в БНФ. На что походит верхний уровень БНФ? Хорошо, это немного зависит от транслируемого языка. Давайте взглянем на Pascal.

СТРУКТУРА ПАСКАЛЯ

Большинство книг по Pascal включают БНФ определение языка. Вот несколько первых строк одного из них:

<program> ::= <program-header> <block> '.'

<program-header> ::= PROGRAM <ident>

<block> ::= <declarations> <statements>

Мы можем написать подпрограммы распознавания для работы с каждым из этих элементов подобно тому, как мы делали это прежде. Для каждого из них мы будем использовать знакомые нам односимвольные токены, затем понемногу расширяя их. Давайте начнем с первого распознавателя: непосредственно программы.

Для ее трансляции мы начнем с новой копии Cradle. Так как мы возвращаемся к односимвольным именам мы будем просто использовать "p" для обозначения «program».

К новой копии Cradle добавьте следующий код и вставьте обращение к нему из основной программы:

```
{--------------------------------------------------------------}
{ Parse and Translate A Program }
procedure Prog;
var Name: char;
begin
Match('p');  {Handles program header part}
Name := GetName;
Prog(Name);
Match('.');
Epilog(Name);
end;
{--------------------------------------------------------------}
```

Процедуры Prog и Epilog выполняют все, что необходимо для связи программы с операционной системой так чтобы она могла выполняться как программа. Само собой разумеется, эта часть будет очень ОС-зависима. Помните, что я выдаю код для 68000, работающий под ОС, которую я использую – SK*DOS. Я понимаю, что большинство из вас использует PC и вы предпочли бы увидеть что-нибудь другое, но я слишком далеко зашел, чтобы что-то сейчас менять!

В любом случае, SK*DOS особенно простая для общения операционная система. Вот код для Prog и Epilog:

```
{--------------------------------------------------------------}
{ Write the Prolog }
procedure Prolog;
begin
EmitLn('WARMST EQU $A01E');
end;
{--------------------------------------------------------------}
{ Write the Epilog }
procedure Epilog(Name: char);
begin
EmitLn('DC WARMST');
EmitLn('END ' + Name);
end;
{--------------------------------------------------------------}
```

Как обычно добавьте этот код и испытайте «компилятор». В настоящее время

существует только одна допустимая входная последовательность:

px. (где x – это любая одиночная буква, имя программы).

Хорошо, как обычно наша первая попытка не очень впечатляет, но я уверен к настоящему времени вы знаете, что дальше станет интересней. Есть одна важная вещь, которую следует отметить: на выходе получается работающая, законченная и выполнимая программа (по крайней мере после того, как она будет ассемблирована).

Это очень важно. Приятная особенность нисходящего метода состоит в том, что на любом этапе вы можете компилировать подмножество завершенного языка и получить программу, которая будет работать на конечной машине. Отсюда, затем, нам необходимо только добавлять возможности, расширяя конструкции языка. Это очень похоже на то, что мы уже делали, за исключением того, что мы подходили к этому с другого конца.

РАСШИРЕНИЕ

Чтобы расширить компилятор мы должны просто работать с возможностями языка последовательно. Я хочу начать с пустой процедуры, которая ничего не делает, затем добавлять детали в пошаговом режиме. Давайте начнем с обработки блока в соответствии с его PDL выше. Мы можем сделать это в два этапа. Сначала добавьте пустую процедуру:

```
–
 Parse and Translate a Pascal Block
procedure DoBlock(Name: char);
begin
end;
–
```

и измените Prog следующим образом:

```
–
 Parse and Translate A Program
procedure Prog;
var Name: char;
begin
Match('p');
Name := GetName;
Prolog;
DoBlock(Name);
Match('.');
Epilog(Name);
end;
–
```

Это конечно не должно изменить поведения программы, и не меняет. Но сейчас определение Prog закончено и мы можем перейти к расширению DoBlock. Это получается прямо из его БНФ определения:

```
–
 Parse and Translate a Pascal Block
procedure DoBlock(Name: char);
begin
Declarations;
PostLabel(Name);
Statements;
end;
–
```

Процедура PostLabel была определена в главе по ветвлениям. Скопируйте ее в вашу

копию Cradle.

Я возможно должен объяснить причину вставки метки. Это имеет отношение к работе SK*DOS. В отличие от некоторых других ОС, SK*DOS позволяет точке входа в основную программу находиться в любом месте программы. Все, что вы должны сделать, это дать этой точке имя. Вызов PostLabel помещает это имя как раз перед первым выполнимым утверждением в основной программе. Как SK*DOS узнает какая из множества меток является точкой входа, спросите вы? Та, которая соответствует утверждению END в конце программы.

Теперь нам нужны заглушки для процедур Declarations и Statements. Сделайте их пустыми процедурами как мы делали это раньше.

Программа все еще делает то же самое? Тогда мы можем перейти к следующему этапу.

ОБЪЯВЛЕНИЯ

БНФ для объявлений в Pascal такая:
<declarations> ::= (<label list> |
<constant list> |
<type list> |
<variable list> |
<procedure> |
<function>)*

(Заметьте, что я использую более либеральное определение, используемое в Turbo Pascal. В определении стандартного Pascal каждая из этих частей должна следовать в определенном порядке относительно других).

Как обычно давайте позволим одиночным символам представлять каждый из этих типов объявлений. Новая форма для Declarations:

```
{
 Parse and Translate the Declaration Part
procedure Declarations;
begin
while Look in ['l', 'c', 't', 'v', 'p', 'f'] do
case Look of
'l': Labels;
'c': Constants;
't': Types;
'v': Variables;
'p': DoProcedure;
'f': DoFunction;
end;
end;
}
```

Конечно, нам нужны процедуры-заглушки для каждого из этих типов объявлений. На этот раз они не могут быть совсем пустыми процедурами, так как иначе мы останемся с бесконечным циклом While. По крайней мере каждая подпрограмма распознавания должна съедать символ, который вызывает ее. Вставьте следующие процедуры:

```
{
 Process Label Statement
procedure Labels;
begin
Match('l');
end;
```

```
_
 Process Const Statement
procedure Constants;
begin
Match('c');
end;
_
 Process Type Statement
procedure Types;
begin
Match('t');
end;
_
 Process Var Statement
procedure Variables;
begin
Match('v');
end;
_
 Process Procedure Definition
procedure DoProcedure;
begin
Match('p');
end;
_
 Process Function Definition
procedure DoFunction;
begin
Match('f');
end;
_
```

Теперь испытайте компилятор используя несколько характерных входных последовательностей. Вы можете смешивать объявления любым образом, каким вам нравится пока последним символом в программе не будет ".", указывающий на конец программы. Конечно, ни одно из этих объявлений фактически ничего не объявляет, так что вам не нужны (и вы не можете использовать) любые символы, кроме тех, которые обозначают ключевые слова.

Мы можем расширить раздел операторов аналогичным способом. БНФ для него будет:

<statements> ::= <compound statement>

<compound statement> ::= BEGIN <statement>(';' <statement>) END

Заметьте, что утверждение может начинаться с любого идентификатора, исключая END. Так что первая пустой формой процедуры Statements будет:

```
_
 Parse and Translate the Statement Part
procedure Statements;
begin
Match('b');
while Look &lt;&gt; 'e' do
GetChar;
Match('e');
end;
_
```

Сейчас компилятор примет любое число объявлений, сопровождаемое блоком BEGIN

основной программы. Сам этот блок может содержать любые символы (за исключением END), но они должны присутствовать.

Простейшая входная форма сейчас

'pxbe.'

Испытайте ее. Также попробуйте некоторые ее комбинации. Сделайте некоторые преднамеренные ошибки и посмотрите что произойдет.

К этому моменту вы должны начать видеть основную линию. Мы начинаем с пустого транслятора для обработки программы, затем в свою очередь мы расширяем каждую процедуру, основанную на ее БНФ определении. Подобно тому, как более низкоуровневые БНФ определения добавляют детали и развивают определения более высокого уровня, более низкоуровневые распознаватели будут анализировать более детально входную программу. Когда последняя заглушка будет расширена, компилятор будет закончен. Это нисходящая разработка/реализация в ее чистейшей форме.

Вы могли бы заметить, что даже хотя мы и добавляли процедуры, выходной результат программы не изменялся. Так и должно быть. На этих верхних уровнях не требуется никакой выдачи кода. Распознаватели функционируют просто как распознаватели. Они принимают входные последовательности, отлавливают плохие и направляют хорошие в нужные места, так что они делают свою работу. Если бы мы занимались этим немного дольше, код начал бы появляться.

Следующим шагом в нашем расширении должна возможно быть процедура Statements. Определение Pascal:

<statement> ::= <simple statement> | <structured statement>
<simple statement> ::= <assignment> | <procedure call> | null
<structured statement> ::= <compound statement> |
<if statement> |
<case statement> |
<while statement> |
<repeat statement> |
<for statement> |
<with statement>

Это начинает выглядеть знакомыми. Фактически вы уже прошли через процесс синтаксического анализа и генерации кода и для операций присваивания и для управляющих структур. Это место, где верхний уровень встречается с нашим восходящим методом из предыдущих уроков. Конструкции будут немного отличаться от тех, которые мы использовали для KISS, но в этих различиях нет ничего, чего бы вы не смогли сделать.

Я думаю теперь вы можете получить представление об этом процессе. Мы начали с завершенного БНФ описания языка. Начиная с верхнего уровня мы закодировали распознаватели для этих БНФ утверждений используя процедуры-заглушки для распознавателей следующего уровня. Затем мы расширили более низкоуровневые утверждения один за другим.

Как оказывается, определение Pascal очень совместимо с использованием БНФ и БНФ описания этого языка существуют в избытке. Вооружившись таким описанием вы обнаружите, что довольно просто продолжить процесс, который мы начали.

Вы могли бы продолжить расширение этих конструкций, просто чтобы прочувствовать это. Я не ожидаю, что вы сможет завершить сейчас компилятор Паскаля... есть слишком много вещей таких как процедуры и типы, к которым мы еще не обращались... но могло бы быть полезным попробовать некоторые из более знакомых вещей. Вам было бы полезно увидеть выполнимые программы, появляющиеся с другого конца.

Если бы я собирался обратиться к вопросам которые мы еще не охватили, я предпочел бы сделать это в контексте KISS. Мы не пытаемся построить полный копилятор Pascal, поэтому я собираюсь остановить на этом расширение Pascal. Давайте взглянем на очень отличающийся язык.

СТРУКТУРА СИ

Язык С совсем другой вопрос, как вы увидите. Книги по С редко включают БНФ определения языка. Возможно дело в том, что этот язык очень сложен для описания в БНФ.

Одна из причин что я показываю вам сейчас эти структуры в том что я могу впечатлить вас двумя фактами:

1. Определение языка управляет структурой компилятора. Что работает для одного языка может быть бедствием для другого. Это очень плохая идея попытаться принудительно встроить данную структуру в компилятор. Скорее вы должны позволить БНФ управлять структурой, как мы делали здесь.

2. Язык, для которого сложно написать БНФ также будет возможно сложен для написания компилятора. Си – популярный язык и он имеет репутацию как позволяющий сделать практически все, что возможно. Несмотря на успех Small C, C является непростым для анализа языком.

Программа на С имеет меньше структур, чем ее аналог на Pascal. На верхнем уровне все в С является статическим объявлением или данных или функций. Мы можем зафиксировать эту мысль так:

<program> ::= (<global declaration>)*

<global declaration> ::= <data declaration> | <function>

В Small C функции могут иметь только тип по умолчанию int, который не объявлен. Это делает входную программу легкой для синтаксического анализа: первым токеном является или «int», «char» или имя функции. В Small C команды препроцессора также обрабатываются компилятором соответствующе, так что синтаксис становится:

<global declaration> ::= '#' <preprocessor command> |

'int' <data list> |

'char' <data list> |

<ident> <function body> |

Хотя мы в действительности больше заинтересованы здесь в полном С , я покажу вам код, соответствующий структуре верхнего уровня Small C.

```
–
 Parse and Translate A Program
procedure Prog;
begin
while Look &lt;&gt; ^Z do begin
case Look of
'#': PreProc;
'i': IntDecl;
'c': CharDecl;
else DoFunction(Int);
end;
end;
end;
–
```

Обратите внимание, что я должен был использовать ^Z чтобы указать на конец исходного кода. С не имеет ключевого слова типа END или "." для индикации конца программы.

С полным Си все не так просто. Проблема возникает потому, что в полном Си функции могут также иметь типы. Так что когда компилятор видит ключевое слово типа «int» он все еще не знает ожидать ли объявления данных или определение функции. Дела становятся более сложными так как следующим токеном может быть не имя... он может начинаться с

"*" или "(" или комбинаций этих двух.

Точнее говоря, БНФ для полного Си начинается с:

<program> ::= (<top-level decl>)*

<top-level decl> ::= <function def> | <data decl>

<data decl> ::= [<class>] <type> <decl-list>

<function def> ::= [<class>] [<type>] <function decl>

Теперь вы можете увидеть проблему: первые две части обьявлений для данных и функций могут быть одинаковыми. Из-за неоднозначности в этой грамматике выше, она является неподходящей для рекурсивного синтаксического анализатора. Можем ли мы преобразовать ее в одну из подходящих? Да, с небольшой работой. Предположим мы запишем ее таким образом:

<top-level decl> ::= [<class>] <decl>

<decl> ::= <type> <typed decl> | <function decl>

<typed decl> ::= <data list> | <function decl>

Мы можем написать подпрограмму синтаксичесого анализа для определений классов и типов и позволять им отложить их сведения и продолжать выполнение даже не зная обрабатывается ли функция или объявление данных.

Для начала, наберите следующую версию основной программы:

```
_
 Main Program
begin
Init;
while Look &lt;&gt; ^Z do begin
GetClass;
GetType;
TopDecl;
end;
end.
_
```

На первый раз просто сделайте три процедуры-заглушки которые ничего не делают, а только вызывают GetChar.

Работает ли эта программа? Хорошо, было бы трудно не сделать это, так как мы в действительности не требовали от нее какой-либо работы. Уже говорилось, что компилятор Си примет практически все без отказа. Несомненно это правда для этого компилятора, потому что в действительности все, что он делает, это съедает входные символы до тех пор, пока не найдет ^Z.

Затем давайте заставим GetClass делать что-нибудь стоящее. Объявите глобальную переменную

var Class: char;

и измените GetClass

```
_
 Get a Storage Class Specifier
Procedure GetClass;
begin
if Look in ['a', 'x', 's'] then begin
Class := Look;
GetChar;
end
else Class := 'a';
end;
_
```

Здесь я использовал три одиночных символа для представления трех классов памяти «auto», «extern» и «static». Это не единственные три возможных класса... есть также «register» и «typedef», но это должно дать вам представление. Заметьте, что класс по умолчанию «auto».

Мы можем сделать подобную вещь для типов. Введите следующую процедуру:

```
{
 Get a Type Specifier
}
procedure GetType;
begin
Typ := ' ';
if Look = 'u' then begin
Sign := 'u';
Typ := 'i';
GetChar;
end
else Sign := 's';
if Look in ['i', 'l', 'c'] then begin
Typ := Look;
GetChar;
end;
end;
{--------------------------------------------------------------}
```

Обратите внимание, что вы должны добавить еще две глобальные переменные Sign и Typ.

С этими двумя процедурами компилятор будет обрабатывать определение классов и типов и сохранять их результаты. Мы можем сейчас обрабатывать остальные объявления.

Мы еще ни коим образом не выбрались из леса, потому что все еще существуют много сложностей только в определении типов до того, как мы дойдем даже до фактических данных или имен функций. Давайте притворимся на мгновение, что мы прошли все эти заслоны и следующим во входном потоке является имя. Если имя сопровождается левой скобкой, то мы имеем объявление функции. Если нет, то мы имеем по крайней мере один элемент данных, и возможно список, каждый элемент которого может иметь инициализатор.

Вставьте следующую версию TopDecl:

```
{--------------------------------------------------------------}
{ Process a Top-Level Declaration }
procedure TopDecl;
var Name: char;
begin
Name := Getname;
if Look = '(' then
DoFunc(Name)
else
DoData(Name);
end;
{--------------------------------------------------------------}
```

(Заметьте, что так как мы уже прочитали имя, мы должны передать его соответствующей подпрограмме.)

Наконец, добавьте две процедуры DoFunc и DoData:

```
{--------------------------------------------------------------}
```

```
 Process a Function Definition
procedure DoFunc(n: char);
begin
Match('(');
Match(')');
Match('');
Match('');
if Typ = ' ' then Typ := 'i';
Writeln(Class, Sign, Typ, ' function ', n);
end;
_
 Process a Data Declaration
procedure DoData(n: char);
begin
if Typ = ' ' then Expected('Type declaration');
Writeln(Class, Sign, Typ, ' data ', n);
while Look = ',' do begin
Match(',');
n := GetName;
WriteLn(Class, Sign, Typ, ' data ', n);
end;
Match(';');
end;
_
```

Так как мы еще далеки от получения выполнимого кода, я решил чтобы эти две подпрограммы только сообщали нам, что они нашли.

Протестируйте эту программу. Для объявления данных дайте список, разделенный запятыми. Мы не можем пока еще обрабатывать инициализаторы. Мы также не можем обрабатывать списки параметров функций но символы «()» должны быть.

Мы все еще очень далеко от того, чтобы иметь компилятор C, но то что у нас есть обрабатывает правильные виды входных данных и распознает и хорошие и плохие входных данные. В процессе этого естественная структура компилятора начинает принимать форму.

Можем ли мы продолжать пока не получим что-то, что действует более похоже на компилятор. Конечно мы можем. Должны ли мы? Это другой вопрос. Я не знаю как вы, но у меня начинает кружиться голова, а мы все еще далеки от того, чтобы даже получить что-то кроме объявления данных.

К этому моменту, я думаю, вы можете видеть как структура компилятора развивается из определения языка. Структуры, которые мы увидели для наших двух примеров, Pascal и C, отличаются как день и ночь. Pascal был разработан, по крайней мере частично, чтобы быть легким для синтаксического анализа и это отразилось в компиляторе. Вообще, Pascal более структурирован и мы имеем более конкретные идеи какие виды конструкций ожидать в любой точке. В C наоборот, программа по существу является списком объявлений завершаемых только концом файла.

Мы могли бы развивать обе эти структуры намного дальше, но помните, что наша цель здесь не в том, чтобы построить компилятор C или Pascal, а скорее изучать компиляторы вообще. Для тех из вас, кто хотят иметь дело с Pascal или C, я надеюсь, что дал вам достаточно начал чтобы вы могли взять их отсюда (хотя вам скоро понадобятся некоторые вещи, которые мы еще не охватили здесь, такие как типы и вызовы процедур). Остальные будьте со мной в следующей главе. Там я проведу вас через разработку законченного компилятора для TINY, подмножества KISS.

Увидимся.

Представление «TINY»

ВВЕДЕНИЕ

В последней главе я показал вам основную идею нисходящей разработки компилятора. Я показал вам первые несколько шагов этого процесса для компиляторов Pascal и C, но я остановился далеко от его завершения. Причина была проста: если мы собираемся построить настоящий, функциональный компилятор для какого-нибудь языка, я предпочел бы сделать это для KISS, языка, который я определил в этой обучающей серии.

В этой главе мы собираемся сделать это же для подмножества KISS, которое я решил назвать TINY.

Этот процесс по существу будет аналогичен выделенному в главе 9, за исключением одного заметного различия. В той главе я предложил вам начать с полного БНФ описания языка. Это было бы прекрасно для какого-нибудь языка типа Pascal или C, определения которого устоялись. В случае же с TINY, однако, мы еще не имеем полного описания... мы будем определять язык по ходу дела. Это нормально. Фактически, это предпочительней, так как мы можем немного подстраивать язык по ходу дела для сохранения простоты анализа.

Так что в последующей разработке мы фактически будем выполнять нисходящую разработку и языка и его компилятора. БНФ описание будет расти вместе с компилятором.

В ходе этого будет принят ряд решений, каждое из которых будет влиять на БНФ и, следовательно, характер языка. В каждой решающей точке я попытаюсь не забывать объяснять решение и разумное обоснование своего выбора. Если вам случится придерживаться другого мнения и вы предпочтете другой вариант, вы можете пойти своим путем. Сейчас вы имеет базу для этого. Я полагаю важно отметить, что ничего из того, что мы здесь делаем не подчинено каким-либо жесткими правилами. Когда вы разрабатываете свой язык вы не должны стесняться делать это своим способом.

Многие из вас могут сейчас спросить: зачем нужно начинать с самого начала? У нас есть работающее подмножество KISS как результат главы 7 (лексический анализ). Почему бы просто не раширить его как нужно? Ответ тройной. Прежде всего, я сделал несколько изменений для упрощения программы... типа изоляции процедур генерации кода, в результате чего мы можем более легко выполнять преобразование для различных машин. Во-вторых, я хочу, чтобы вы увидели что разработка действительно может быть выполнена сверху вниз как это подчеркнуто в последней главе. Наконец, нам всем нужна практика. Каждый раз, когда я прохожу через эти упражнения, я начинаю понимать немного больше, и вы будете тоже.

ПОДГОТОВКА

Много лет назад существовали языки, называемые Tiny BASIC, Tiny Pascal и Tiny C, каждый из которых был подмножеством своего полного родительского языка. Tiny BASIC, к примеру, имел только односимвольные имена переменных и глобальные переменные. Он поддерживал только один тип данных. Звучит знакомо? К этому моменту мы имеем почти все инструменты, необходимые для создания компилятора подобного этому.

Однако язык, называемый Tiny-такой-то все же несет некоторый багаж, унаследованный от своего родительского языка. Я часто задавался вопросом, хорошая ли это идея. Согласен, язык, основанный на каком-то родительском языке, будет иметь преимущество знакомости, но может также существовать некоторый особенный синтаксис, перенесенный из родительского языка, который может приводить к появлению ненужной сложности в компиляторе. (Нигде это не является большей истиной, чем в Small C).

Я задавался вопросом, насколько маленьким и простым может быть создан компилятор и при этом все еще быть полезным, если он разрабатывался из условия быть легким и для использования и для синтаксического анализа. Давайте выясним. Этот язык будет

называться просто «TINY». Он является подмножеством KISS, который я также еще полностью не определил, что по крайней мере делает нас последовательными (!). Я полагаю вы могли бы назвать его TINY KISS. Но это открывает целую кучу проблем, так что давайте просто придерживаться имени TINY.

Главные ограничения TINY будут возникать из-за тех вещей, которые мы еще не рассмотрели, таких как типы данных. Подобно своим кузенам Tiny C и Tiny BASIC, TINY будет иметь только один тип данных, 16-разрядное целое число. Первая версия, которую мы разработаем, не будет также иметь вызовов процедур и будет использовать односимвольные имена переменных, хотя, как вы увидите, мы можем удалить эти ограничения без особых усилий.

Язык, который я придумал, разделит некоторые хорошие особенности Pascal, C и Ada. Получив урок из сравнения компиляторов Pascal и C в предыдущей главе, TINY все же будет иметь преимущественно вкус Паскаля. Везде, где возможно, структура языка будет ограничена ключевыми словами или символами, так что синтаксический анализатор будет знать, что происходит без догадок.

Другое основное правило: Я хотел бы чтобы в течение всей разработки компилятор производил настоящий выполнимый код. Даже если его не может быть слишком много в самом начале, но по крайней мере он должен быть корректным.

Наконец, я буду использовать пару ограничений Pascal, которые имеют смысл: Все данные и процедуры должны быть объявлены перед тем, как они используются. Это имеет большой смысл, даже если сейчас единственным типом данных, который мы будем использовать, будет слово. Это правило, в свою очередь, означает, что единственное приемлемое место для размещения выполнимого кода основной программы – в конце листинга.

Определение верхнего уровня будет аналогично Pascal:

<program> ::= PROGRAM <top-level decl> <main> '.'

Мы уже достигли решающей точки. Моей первой мыслью было сделать основной блок необязательным. Кажется бессмысленным писать «программу» без основной программы, но это имеет смысл, если мы разрешим множественные модули, связанные вместе. Фактически я предполагаю учесть это в KISS. Но тогда мы столкнемся с кучей проблем, которые я предпочел бы сейчас не затрагивать. Например, термин «PROGRAM» в действительности становится неправильно употребляемым. MODULE из Modula-2 или UNIT из Turbo Pascal были бы более подходящими. Во-вторых, как насчет правил видимости? Нам необходимо соглашение для работы с видимостью имен в модулях. На данный момент лучше просто сохранить простоту и совершенно игнорировать эту идею.

Также необходимо определиться с требованием, чтобы основная программа была последней. Я играл с идеей сделать ее размещение нефиксированным как в C. Характер SK*DOS, ОС под которую я компилирую, позволяет сделать это очень просто. Но это в действительности не имеет большого смысла принимая во внимание Pascal-подобное требование, что все данные и процедуры должны быть обьявлены прежде чем они используются. Так как основная программа может вызывать только те процедуры, которые уже были объявлены, единственное местоположение, имеющее смысл – в конце, a la Pascal.

По данной выше БНФ давайте напишем синтаксический анализатор, который просто распознает скобки:

```
_
 Parse and Translate a Program
procedure Prog;
begin
Match('p');
Header;
Prolog;
Match('.');
```

```
Epilog;
end;
–
```

Процедура Header просто выдает инициализационный код, необходимый ассемблеру:

```
–
 Write Header Info
procedure Header;
begin
WriteLn('WARMST', TAB, 'EQU $A01E');
end;
–
```

Процедуры Prolog и Epilog выдают код для идентификации основной программы и для возвращения в ОС:

```
–
 Write the Prolog
procedure Prolog;
begin
PostLabel('MAIN');
end;
–
 Write the Epilog
procedure Epilog;
begin
EmitLn('DC WARMST');
EmitLn('END MAIN');
end;
–
```

Основная программа просто вызывает Prog и затем выполняет проверку на чистое завершение:

```
–
 Main Program
begin
Init;
Prog;
if Look &lt;&gt; CR then Abort('Unexpected data after "."');
end.
–
```

Сейчас TINY примет только одну «программу» – пустую:

PROGRAM . (или 'p.' в нашей стенографии).

Заметьте, тем не менее, что компилятор генерирует для этой программы корректный код. Она будет выполняться и делать то, что можно ожидать от пустой программы, т.е. ничего кроме элегантного возвращения в ОС.

Один из моих любимых бенчмарков для компиляторов заключается в компиляции, связывании и выполнении пустой программы для любого языка. Вы можете многое узнать о реализации измеряя предел времени, необходимый для компиляции тривиальной программы. Также интересно измерить количество полученного кода. Во многих компиляторах код может быть довольно большим, потому что они всегда включают целую run-time библиотеку независимо от того, нуждаются они в ней или нет. Ранние версии Turbo Pascal в этом случае

производили объектный файл 12К. VAX C генерирует 50К!

Самые маленькие пустые программы какие я видел, получены компиляторами Модула-2 и они занимают примерно 200-800 байт.

В случае TINY у нас еще нет run-time библиотеки, так что объектный код действительно крошечный (tiny): два байта. Это стало рекордом, и вероятно останется таковым, так как это минимальный размер, требуемый ОС.

Следующим шагом будет обработка кода для основной программы. Я буду использовать блок BEGIN из Pascal:

<main> ::= BEGIN <block> END

Здесь мы снова приняли решение. Мы могли бы потребовать использовать объявление вида «PROCEDURE MAIN», подобно C. Я должен допустить, что это совсем неплохая идея... Мне не особенно нравится подход Паскаля так как я предпочитаю не иметь проблем с определением местоположения основной программы в листинге Паскаля. Но альтернатива тоже немного неудобна, так как вы должны работать с проверкой ошибок когда пользователь опустит основную программу или сделает орфографическую ошибку в ее названии. Здесь я использую простой выход.

Другое решение проблемы «где расположена основная программа» может заключаться в требовании имени для программы и заключения основной программы в скобки:

BEGIN <name>

END <name>

аналогично соглашению Модула-2. Это добавляет в язык немного «синтаксического сахара». Подобные вещи легко добавлять и изменять по вашим симпатиям если вы сами проектируете язык.

Для синтаксического анализа такого определения основного блока измените процедуру Prog следующим образом:

```
_
 Parse and Translate a Program
procedure Prog;
begin
Match('p');
Header;
Main;
Match('.');
end;
_
```

и добавьте новую процедуру:

```
_
 Parse and Translate a Main Program
procedure Main;
begin
Match('b');
Prolog;
Match('e');
Epilog;
end;
_
```

Теперь единственной допустимой программой является программа:

PROGRAM BEGIN END. (или 'pbe.')

Разве мы не делаем успехи??? Хорошо, как обычно это становится лучше. Вы могли бы попробовать сделать здесь некоторые преднамеренные ошибки подобные пропуску 'b' или 'e'

и посмотреть что случится. Как всегда компилятор должен отметить все недопустимые входные символы.

ОБЪЯВЛЕНИЯ

Очевидно на следующем шаге необходимо решить, что мы подразумеваем под объявлением. Я намереваюсь иметь два вида объявлений: переменных и процедур/функций. На верхнем уровне разрешены только глобальные объявления, точно как в C.

Сейчас здесь могут быть только объявления переменных, идентифицируемые по ключевому слову VAR (сокращенно "v").

<top-level decls> ::= (<data declaration>)*

<data declaration> ::= VAR <var-list>

Обратите внимание, что так как имеется только один тип переменных, нет необходимости объявлять этот тип. Позднее, для полной версии KISS, мы сможем легко добавить описание типа.

Процедура Prog становится:

```
{
 Parse and Translate a Program
procedure Prog;
begin
Match('p');
Header;
TopDecls;
Main;
Match('.');
end;
}
```

Теперь добавьте две новые процедуры:

```
{
 Process a Data Declaration
procedure Decl;
begin
Match('v');
GetChar;
end;
{
 Parse and Translate Global Declarations
procedure TopDecls;
begin
while Look &amp;lt;&amp;gt; 'b' do
case Look of
'v': Decl;
else Abort('Unrecognized Keyword "' + Look + '"');
end;
end;
}
```

Заметьте, что на данный момент Decl – просто заглушка. Она не генерирует никакого кода и не обрабатывает список... каждая переменная должна быть в отдельном утверждении VAR.

ОК, теперь у нас может быть любое число объявлений данных, каждое начинается с "v" вместо VAR, перед блоком BEGIN. Попробуйте несколько вариантов и посмотрите, что

происходит.

ОБЪЯВЛЕНИЯ И ИДЕНТИФИКАТОРЫ

Это выглядит довольно хорошо, но мы все еще генерируем только пустую программу. Настоящий ассемблер должен выдавать директивы ассемблера для распределения памяти под переменные. Пришло время действительно получить какой-нибудь код.

С небольшим дополнительным кодом это легко сделать в процедуре Decl. Измените ее следующим образом:

```
{
 Parse and Translate a Data Declaration
procedure Decl;
var Name: char;
begin
Match('v');
Alloc(GetName);
end;
}
```

Процедура Alloc просто выдает команду ассемблеру для распределения памяти:

```
{
 Allocate Storage for a Variable
procedure Alloc(N: char);
begin
WriteLn(N, ':', TAB, 'DC 0');
end;
}
```

Погоняйте программу. Попробуйте входную последовательность, которая объявляет какие-нибудь переменные, например:

pvxvyvzbe.

Видите, как распределяется память? Просто, да? Заметьте также, что точка входа «MAIN» появляется в правильном месте.

Кстати, «настоящий» компилятор имел бы также таблицу идентификаторов для записи используемых переменных. Обычно, таблица идентификаторов необходима для записи типа каждой переменной. Но так как в нашем случае все переменные имеют один и тот же тип, нам не нужна таблица идентификаторов. Оказывается, мы смогли бы находить идентификатор даже без различия типов, но давайте отложим это пока не возникнет такая необходимость.

Конечно, в действительности мы не анализировали правильный синтаксис для объявления данных, так как он включает список переменных. Наша версия разрешает только одну переменную. Это также легко исправить.

БНФ для <var-list> следующая:

<var-list> ::= <ident> (, <ident>)*

Добавление этого синтаксиса в Decl дает новую версию:

```
{
 Parse and Translate a Data Declaration
procedure Decl;
var Name: char;
begin
Match('v');
```

```
Alloc(GetName);
while Look = ',' do begin
GetChar;
Alloc(GetName);
end;
end;
_
```

ОК, теперь откомпилируйте этот код и испытайте его. Попробуйте ряд строк с объявлениями VAR, попробуйте список из нескольких переменных в одной строке и комбинации этих двух. Работает?

ИНИЦИАЛИЗАТОРЫ

Пока мы работали с объявлениями данных, меня беспокоила одна вещь – то, что Pascal не позволяет инициализировать данные в объявлении. Эта возможность по общему признанию является своего рода излишеством, и ее может не быть в языке, который считается минимальным языком. Но ее также настолько просто добавить, что было бы позором не сделать этого. БНФ становится:

<var-list> ::= <var> (<var>)*

<var> ::= <ident> [= <integer>]

Измените Alloc как показано ниже:

```
_
 Allocate Storage for a Variable
procedure Alloc(N: char);
begin
Write(N, ':', TAB, 'DC ');
if Look = '=' then begin
Match('=');
WriteLn(GetNum);
end
else
WriteLn('0');
end;
_
```

Вот оно: инициализатор в шесть дополнительных строк Pascal.

Испытайте эту версию TINY и проверьте, что вы действительно можете задавать начальное значение перменных.

Ей богу, он начинает походить на настоящий компилятор! Конечно, он все еще ничего не делает, но выглядит хорошо, не так ли?

Перед тем как оставить этот раздел я должен подчеркнуть, что мы использовали две версии GetNum. Одна, более ранняя, возвращала символьное значение, одиночную цифру. Другая принимала многозначное целое число и возвращала целочисленное значение. Любая из них будет работать здесь, так как WriteLn поддерживает оба типа. Но нет никакой причины ограничивать себя одноразрядными значениями, так что правильной версией для использования будет та, которая возвращает целое число. Вот она:

```
_
 Get a Number
function GetNum: integer;
var Val: integer;
begin
```

```
Val := 0;
if not IsDigit(Look) then Expected('Integer');
while IsDigit(Look) do begin
Val := 10 * Val + Ord(Look) – Ord('0');
GetChar;
end;
GetNum := Val;
end;
–
```

Строго говоря, мы должны разрешить выражения в поле данных инициализатора, или, по крайней мере, отрицательные значения. Сейчас давайте просто разрешим отрицательные значения изменив код для Alloc следующим образом:

```
–
 Allocate Storage for a Variable
procedure Alloc(N: char);
begin
if InTable(N) then Abort('Duplicate Variable Name ' + N);
ST[N] := 'v';
Write(N, ':', TAB, 'DC ');
if Look = '=' then begin
Match('=');
If Look = '-' then begin
Write(Look);
Match('-');
end;
WriteLn(GetNum);
end
else
WriteLn('0');
end;
–
```

Теперь у вас есть возможность инициализировать переменные отрицательными и/или многозначными значениями.

ТАБЛИЦА ИДЕНТИФИКАТОРОВ

Существует одна проблема с компилятором в его текущем состоянии: он ничего не делает для сохранения переменной когда мы ее объявляем. Так что компилятор совершенно спокойно распределит память для нескольких переменных с тем же самым именем. Вы можете легко убедиться в этом набрав строку типа

pvavavabe.

Здесь мы объявили переменную A три раза. Как вы можете видеть, компилятор бодро принимает это и генерирует три идентичных метки. Не хорошо.

Позднее, когда мы начнем ссылаться на переменные, компилятор также будет позволять нам ссылаться на переменные, которые не существуют. Ассемблер отловит обе эти ошибки, но это совсем не кажется дружественным поведением – передавать такую ошибку ассемблеру. Компилятор должен отлавливать такие вещи на уровне исходного языка.

Так что даже притом, что нам не нужна таблица идентификаторов для записи типов данных, мы должны установить ее только для того, чтобы проверять эти два условия. Так как пока мы все еще ограничены односимвольными именами переменных таблица идентификаторов может быть тривиальной. Чтобы предусмотреть ее сначала добавьте

следующее объявление в начало вашей программы:

var ST: array['A'..'Z'] of char;

и вставьте следующую функцию:

```
{
 Look for Symbol in Table
function InTable(n: char): Boolean;
begin
InTable := ST[n] &lt;&gt; ' ';
end;
}
```

Нам также необходимо инициализировать таблицу пробелами. Следующие строки в Init сделают эту работу:

```
var i: char;
begin
for i := 'A' to 'Z' do
ST[i] := ' ';
...
```

Наконец, вставьте следующие две строки в начало Alloc:

```
if InTable(N) then Abort('Duplicate Variable Name ' + N);
ST[N] := 'v';
```

Это должно все решить. Теперь компилятор будет отлавливать двойные объявления. Позднее мы также сможем использовать InTable при генерации ссылок на переменные.

ВЫПОЛНИМЫЕ УТВЕРЖДЕНИЯ

К этому времени мы можем генерировать пустую программу, которая имеет несколько объявленных переменных и возможно инициализированных. Но пока мы не генерировали ни строки выполнимого кода.

Верите ли вы или нет, но мы почти имеем пригодный для использования компилятор! Отсутствует только выполнимый код, который должен входить в основную программу. Но этот код – это только операции присваивания и операторы управления... все вещи, которые мы сделали раньше. Так что у нас не должно занять слишком много времени предусмотреть также и их.

БНФ определение, данное раньше для основной программы, включало операторный блок, который мы пока что игнорировали:

<main> ::= BEGIN <block> END

Сейчас мы можем рассматривать блок просто как серию операций присваивания:

<block> ::= (Assignment)*

Давайте начнем с добавления синтаксического анализатора для блока. Мы начнем с процедуры-заглушки для операции присваивания:

```
{
 Parse and Translate an Assignment Statement
procedure Assignment;
begin
GetChar;
end;
{
 Parse and Translate a Block of Statements
procedure Block;
begin
while Look &lt;&gt; 'e' do
```

```
Assignment;
end;
_
```

Измените процедуру Main чтобы она вызывала Block как показано ниже:

```
_
 Parse and Translate a Main Program
procedure Main;
begin
Match('b');
Prolog;
Block;
Match('e');
Epilog;
end;
_
```

Эта версия все еще не генерирует никакого кода для «операций присваивания»... все что она делает это съедает символы до тех пор, пока не увидит "e", означающее «END». Но она устанавливает основу для того, что следует дальше.

Следующий шаг, конечно, – это расширение кода для операций присваивания. Это то, что мы делали много раз до этого, поэтому я не буду задерживаться на этом. На этот раз, однако, я хотел бы работать с генерацией кода немного по-другому. До настоящего времени мы всегда просто вставляли Emits, которые генерируют выходной код в соответствии с подпрограммами синтасического анализа. Немного неструктурно, возможно, но это кажется самым простым способом и помогает видеть, какой код должен быть выдан для каждой конструкции.

Однако, я понимаю, что большинство из вас используют компьютер 80x86, так что от кода, сгенерированного для 68000 вам мало пользы. Некоторые из вас спрашивали меня, что если бы машинозависимый код мог бы быть собран в одном месте, то было бы проще перенастроить его на другой ЦПУ. Ответ конечно да.

Чтобы сделать это вставьте следующие подпрограммы «генерации кода»:

```
_
 Clear the Primary Register
procedure Clear;
begin
EmitLn('CLR D0');
end;
_
 Negate the Primary Register
procedure Negate;
begin
EmitLn('NEG D0');
end;
_
 Load a Constant Value to Primary Register
procedure LoadConst(n: integer);
begin
Emit('MOVE #');
WriteLn(n, ',D0');
end;
_
 Load a Variable to Primary Register
```

```
procedure LoadVar(Name: char);
begin
if not InTable(Name) then Undefined(Name);
EmitLn('MOVE ' + Name + '(PC),D0');
end;
_
 Push Primary onto Stack
procedure Push;
begin
EmitLn('MOVE D0,-(SP)');
end;
_
 Add Top of Stack to Primary
procedure PopAdd;
begin
EmitLn('ADD (SP)+,D0');
end;
_
 Subtract Primary from Top of Stack
procedure PopSub;
begin
EmitLn('SUB (SP)+,D0');
EmitLn('NEG D0');
end;
_
 Multiply Top of Stack by Primary
procedure PopMul;
begin
EmitLn('MULS (SP)+,D0');
end;
_
 Divide Top of Stack by Primary
procedure PopDiv;
begin
EmitLn('MOVE (SP)+,D7');
EmitLn('EXT.L D7');
EmitLn('DIVS D0,D7');
EmitLn('MOVE D7,D0');
end;
_
 Store Primary to Variable
procedure Store(Name: char);
begin
if not InTable(Name) then Undefined(Name);
EmitLn('LEA ' + Name + '(PC),A0');
EmitLn('MOVE D0,(A0)')
end;
_
```

Приятная особенность такого подхода, конечно, в том что мы можем перенастроить компилятор на новый ЦПУ просто переписав эти процедуры «генератора кода». Кроме того, позднее мы обнаружим что можем улучшить качество кода немного подправляя эти процедуры без необходимости изменения компилятора.

Обратите внимание, что и LoadVar и Store проверяют таблицу идентификаторов чтобы удостовериться, что переменная определена. Обработчик ошибки Undefined просто вызывает Abort:

```
_
 Report an Undefined Identifier
procedure Undefined(n: string);
begin
Abort('Undefined Identifier ' + n);
end;
_
```

Итак, теперь мы наконец готовы начать обработку выполнимого кода. Мы сделаем это заменив пустую версию процедуры Assignment.

Мы проходили этот путь много раз прежде, так что все это должно быть вам знакомо. Фактически, если бы не изменения, связанные с генерацией кода, мы могли бы просто скопировать процедуры из седьмой части. Так как мы сделали некоторые изменения я не буду их просто копировать, но мы пройдем немного быстрее, чем обычно.

БНФ для операций присваивания:

<assignment> ::= <ident> = <expression>
<expression> ::= <first term> (<addop> <term>)*
<first term> ::= <first factor> <rest>
<term> ::= <factor> <rest>
<rest> ::= (<mulop> <factor>)*
<first factor> ::= [<addop>] <factor>
<factor> ::= <var> | <number> | (<expression>)

Эта БНФ также немного отличается от той, что мы использовали раньше... еще одна «вариация на тему выражений». Эта специфичная версия имеет то, что я считаю лучшей обработкой унарного минуса. Как вы увидите позднее, это позволит нам очень эффективно обрабатывать отрицательные константы. Здесь стоит упомянуть, что мы часто видели преимущества «подстраивания» БНФ по ходу дела, с цель сделать язык легким для анализа. То, что вы видете здесь, немного другое: мы подстраиваем БНФ для того, чтобы сделать генерацию кода более эффективной! Это происходит впервые в этой серии.

Во всяком случае, следующий код реализует эту БНФ:

```
_
 Parse and Translate a Math Factor
procedure Expression; Forward;
procedure Factor;
begin
if Look = '(' then begin
Match('(');
Expression;
Match(')');
end
else if IsAlpha(Look) then
LoadVar(GetName)
else
LoadConst(GetNum);
end;
_
 Parse and Translate a Negative Factor
procedure NegFactor;
begin
Match('-');
if IsDigit(Look) then
LoadConst(-GetNum)
else begin
```

```
Factor;
Negate;
end;
end;
–
 Parse and Translate a Leading Factor
procedure FirstFactor;
begin
case Look of
'+': begin
Match('+');
Factor;
end;
'-': NegFactor;
else Factor;
end;
end;
–
 Recognize and Translate a Multiply
procedure Multiply;
begin
Match('*');
Factor;
PopMul;
end;
–
 Recognize and Translate a Divide
procedure Divide;
begin
Match('/');
Factor;
PopDiv;
end;
–
 Common Code Used by Term and FirstTerm
procedure Term1;
begin
while IsMulop(Look) do begin
Push;
case Look of
'*': Multiply;
'/': Divide;
end;
end;
end;
–
 Parse and Translate a Math Term
procedure Term;
begin
Factor;
Term1;
end;
–
 Parse and Translate a Leading Term
procedure FirstTerm;
begin
FirstFactor;
```

```
Term1;
end;
–
 Recognize and Translate an Add
procedure Add;
begin
Match('+');
Term;
PopAdd;
end;
–
 Recognize and Translate a Subtract
procedure Subtract;
begin
Match('-');
Term;
PopSub;
end;
–
 Parse and Translate an Expression
procedure Expression;
begin
FirstTerm;
while IsAddop(Look) do begin
Push;
case Look of
'+': Add;
'-': Subtract;
end;
end;
end;
–
 Parse and Translate an Assignment Statement
procedure Assignment;
var Name: char;
begin
Name := GetName;
Match('=');
Expression;
Store(Name);
end;
–
```

ОК, если вы вставили весь этот код, тогда откомпилируйте и проверьте его. Вы должны увидеть приемлемо выглядящий код, представляющий собой законченную программу, которая будет ассемблироваться и выполняться. У нас есть компилятор!

БУЛЕВА ЛОГИКА

Следующий шаг также должен быть вам знаком. Мы должны добавить булевы выражения и операторы отношений. Снова, так как мы работали с ними не один раз, я не буду подробно разбирать их за исключением моментов, в которых они отличаются от того, что мы делали прежде. Снова, мы не будем просто копировать их из других файлов потому что я немного изменил некоторые вещи. Большинство изменений просто включают изоляцию машинозависимых частей как мы делали для арифметических операций. Я также несколько изменил процедуру NotFactor для соответствия структуре FirstFactor. Наконец я

исправил ошибку в объектном коде для операторов отношений: в инструкции Scc я использовал только младшие 8 бит D0. Нам нужно установить логическую истину для всех 16 битов поэтому я добавил инструкцию для изменения младшего байта.

Для начала нам понадобятся несколько подпрограмм распознавания:

```
_
 Recognize a Boolean Orop
function IsOrop(c: char): boolean;
begin
IsOrop := c in ['|', '~'];
end;
_
 Recognize a Relop
function IsRelop(c: char): boolean;
begin
IsRelop := c in ['=', '#', '&lt;', '&gt;'];
end;
_
```

Также нам понадобятся несколько подпрограмм генерации кода:

```
_
 Complement the Primary Register
procedure NotIt;
begin
EmitLn('NOT D0');
end;
_
.
.
.
_
 AND Top of Stack with Primary
procedure PopAnd;
begin
EmitLn('AND (SP)+,D0');
end;
_
 OR Top of Stack with Primary
procedure PopOr;
begin
EmitLn('OR (SP)+,D0');
end;
_
 XOR Top of Stack with Primary
procedure PopXor;
begin
EmitLn('EOR (SP)+,D0');
end;
_
 Compare Top of Stack with Primary
procedure PopCompare;
begin
EmitLn('CMP (SP)+,D0');
end;
_
 Set D0 If Compare was =
procedure SetEqual;
begin
```

```
EmitLn('SEQ D0');
EmitLn('EXT D0');
end;
_
 Set D0 If Compare was !=
procedure SetNEqual;
begin
EmitLn('SNE D0');
EmitLn('EXT D0');
end;
_
 Set D0 If Compare was &gt;
procedure SetGreater;
begin
EmitLn('SLT D0');
EmitLn('EXT D0');
end;
_
 Set D0 If Compare was &lt;
procedure SetLess;
begin
EmitLn('SGT D0');
EmitLn('EXT D0');
end;
_
```

Все это дает нам необходимые инструменты. БНФ для булевых выражений такая:

<bool-expr> ::= <bool-term> (<orop> <bool-term>)*

<bool-term> ::= <not-factor> (<andop> <not-factor>)*

<not-factor> ::= ['!'] <relation>

<relation> ::= <expression> [<relop> <expression>]

Зоркие читатели могли бы заметить, что этот синтаксис не включает нетерминал «bool-factor» используемый в ранних версиях. Тогда он был необходим потому, что я также разрешал булевы константы TRUE и FALSE. Но не забудьте, что в TINY нет никакого различия между булевыми и арифметическими типами... они могут свободно смешиваться. Так что нет нужды в этих предопределенных значениях... мы можем просто использовать -1 и 0 соответственно.

В терминологии C мы могли бы всегда использовать определения:

#define TRUE -1

#define FALSE 0

(Так было бы, если бы TINY имел препроцессор.) Позднее, когда мы разрешим объявление констант, эти два значения будут предопределены языком.

Причина того, что я заостряю на этом ваше внимание, в том что я пытался использовать альтернативный путь, который заключался в использовании TRUE и FALSE как ключевых слов. Проблема с этим подходом в том, что он требует лексического анализа каждого имени переменной в каждом выражении. Как вы помните, я указал в главе 7, что это значительно замедляет компилятор. Пока ключевые слова не могут быть в выражениях нам нужно выполнять сканирование только в начале каждого нового оператора... значительное улучшение. Так что использование вышеуказанного синтаксиса не только упрощает синтаксический анализ, но также ускоряет сканирование.

Итак, если мы удовлетворены синтаксисом, представленным выше, то соответствующий код показан ниже:

```
_
 Recognize and Translate a Relational «Equals»
```

```
procedure Equals;
begin
Match('=');
Expression;
PopCompare;
SetEqual;
end;
_
 Recognize and Translate a Relational «Not Equals»
procedure NotEquals;
begin
Match('#');
Expression;
PopCompare;
SetNEqual;
end;
_
 Recognize and Translate a Relational «Less Than»
procedure Less;
begin
Match('&lt;');
Expression;
PopCompare;
SetLess;
end;
_
 Recognize and Translate a Relational «Greater Than»
procedure Greater;
begin
Match('&gt;');
Expression;
PopCompare;
SetGreater;
end;
_
 Parse and Translate a Relation
procedure Relation;
begin
Expression;
if IsRelop(Look) then begin
Push;
case Look of
'=': Equals;
'#': NotEquals;
'&lt;': Less;
'&gt;': Greater;
end;
end;
end;
_
 Parse and Translate a Boolean Factor with Leading NOT
procedure NotFactor;
begin
if Look = '!' then begin
Match('!');
Relation;
NotIt;
```

```
end
else
Relation;
end;
–
 Parse and Translate a Boolean Term
procedure BoolTerm;
begin
NotFactor;
while Look = '&amp;amp;' do begin
Push;
Match('&amp;amp;');
NotFactor;
PopAnd;
end;
end;
–
 Recognize and Translate a Boolean OR
procedure BoolOr;
begin
Match('|');
BoolTerm;
PopOr;
end;
–
 Recognize and Translate an Exclusive Or
procedure BoolXor;
begin
Match('~');
BoolTerm;
PopXor;
end;
–
 Parse and Translate a Boolean Expression
procedure BoolExpression;
begin
BoolTerm;
while IsOrOp(Look) do begin
Push;
case Look of
'|': BoolOr;
'~': BoolXor;
end;
end;
end;
–
```

Чтобы связать все это вместе не забудьте изменить обращение к Expression в процедурах Factor и Assignment на вызов BoolExpression.

Хорошо, если вы набрали все это, откомпилируйте и погоняйте эту версию. Сначала удостоверьтесь, что вы все еще можете анализировать обычные арифметические выражения. Затем попробуйте булевские. Наконец удостоверьтесь, что вы можете присваивать результат сравнения. Попробуйте к примеру:

pvx,y,zbx=z>ye.

что означает

PROGRAM

VAR X,Y,Z
BEGIN
X = Z > Y
END.
Видите как происходит присваивание булевского значения X?

УПРАВЛЯЮЩИЕ СТРУКТУРЫ

Мы почти дома. Имея булевы выражения легко добавить управляющие структуры. Для TINY мы разрешим только две из них, IF и WHILE:

<if> ::= IF <bool-expression> <block> [ELSE <block>] ENDIF

<while> ::= WHILE <bool-expression> <block> ENDWHILE

Еще раз позвольте мне разъяснить решения, подразумевающиеся в этом синтаксисе, который сильно отличается от синтаксиса C или Pascal. В обоих этих языках «тело» IF или WHILE расценивается как одиночный оператор. Если вы предполагаете использовать блок из более чем одного оператора вы должны создать составной утверждение использую BEGIN-END (в Pascal) или " (в C). В TINY (и KISS) нет таких вещей как составное утверждение... одиночное или множественное, они являются в этом языке просто блоками.

В KISS все управляющие структуры имеют явные и уникальные ключевые слова, выделяющие операторный блок поэтому не может быть никакой путаницы где он начинается и заканчивается. Это современный подход, используемый в таких уважаемых языках, как Ada и Modula-2 и он полностью устраняет проблему «висячих else».

Обратите внимание, что я мог бы использовать то же самое ключевое слово END для завершения всех конструкций, как это сделано в Pascal. (Закрывающая '