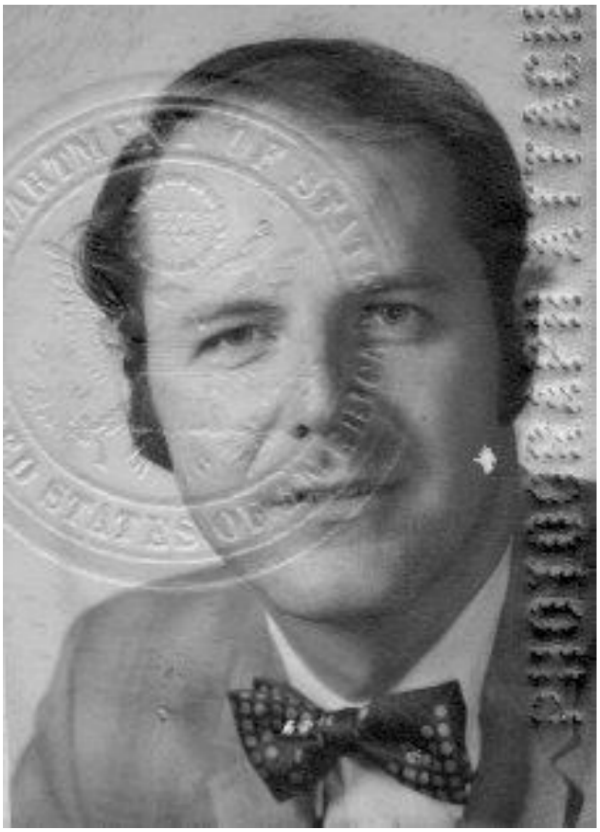
# PROGRAMMING

# A PROBLEM-ORIENTED

# LANGUAGE



**Charles H. Moore**

# 序文

これは、昔書いた未発表の本です。Forthの最初のバージョンを書いた直後です。おそらく、Forthを作った動機を説明しています。今でも興味はあります。しかし、ホームページに掲載できるので、出版はもう関係ありません。

昔、Forth, Incから回収した原稿があります。Smith-Coronaポータブルで打ったもので、オーバーストライクと注釈もついています。読みにくいので、HTMLで再入力しています。

この打ち込み作業は役に立ちます。昔はタイピングが得意だったのですが、今は衰えてしまったので、指の運動にもなるし、エラー率を減らせるかどうかも気になる。

テキストには最小限の変更しか加えず、HTMLに合わせるだけです。古風な言葉、文法的でない言葉、不明瞭な言葉はそのままに。残った誤字脱字はすべて現代のものです。

Chuck Moore 2011



## PROGRAMMING A PROBLEM-ORIENTED LANGUAGE

## Charles H. Moore written ~ June 1970

# 1. Introduction

あなたがなぜこの本を読んでいるのか、よくわからない。なぜこの本を書いているのか、それを発見するのに時間がかかった。タイトルを見てみましょう。「問題指向言語のプログラミング」。キーワードは「プログラミング」です。私はこれまで多くのプログラムを書いてきた。良いプログラムを書こうと努力し、その書き方を批判的に観察してきた。そして、より少ない労力で、より質の高いプログラムを作ることを目指してきた。

このような観察の過程で、私は自分が同じ間違いを繰り返していることに気づきました。振り返ってみれば一目瞭然なのに、現場にいたときにはなかなか気づかないミスです。プログラミングの処方箋を書けば、少なくとも問題点を思い出すことができると思ったのです。そして、その結果が私にとって価値があるものであれば、他の人にとっても価値があるはずです。私が言うことがあなたにとって新しいものであれば、価値あるものを学べるかもしれませんし、見慣れた分野を扱っていたとしても、少なくとも新しい視点を得ることができるはずです。

また、私が重要だと考える問題に対して、他の人が関心を示さないことにも心を痛めています。自分のプログラムはそこそこ良いものだ、必要にして十分なものだ、という気軽な態度が、品質に対する一般的な無関心につながるのです。私は、この自信は見当違いであると確信しています。さらに、このような態度は、高級言語への大きな流れと、その非効率性を平然と受け入れることによっても強化されている。本当に優れたアルゴリズムを設計しても、コンパイラがそれを台無しにしてしまうのであれば、何の意味があるのでしょうか?

そこで、私はプログラミングについての本を書きました。私は一方通行の通信回線で議論する趣味はないし、私の言うことが正しいということを納得させることにもあまり興味がない。だから、私が無愛想であることに気がつくだろう。そのため、あなたが疑問に思うようなことをズバッと言ってしまう可能性があります。どうぞ、ご容赦ください。私の意図は、私が有用と感じたアプローチを記録し、おそらくプログラミングへの批判的な関心を刺激することです。もし、あなたが問題を提起するほど気に掛けてくれるなら、私は嬉しく思います。

タイトルに戻ろう。問題指向言語(Problem-Oriented-Language)とは? 私はそれについて書くために始めたわけではありませんし、そうする資格があるかどうかもわかりません。しかし、自分がやっていることを正当化し、それを行うための適切な状況を特定するためには、この言葉が不可欠であることを発見したのです。

問題志向型言語とは、特定の応用(アプリケーション)に合わせた言語のことである。この独特の不器用な用語を避けるため、私は通常、アプリケーション言語(application language)と置き換えることにしています。このような言語は、それが何であるかが認識されていないことが非常に多いのです。たとえば、あなたのプログラムが80列目のコードを読んで入力カードを識別する場合、あなたはアプリケーション言語を実装していることになります。非常に粗雑なもの、非常に厄介なものですが、それはあなたがこの問題を全く考えていなかったからです。問題を認識すれば、より良い解決策を設計することができるはずです。本書はその方法を紹介します。

## 1.1 基本原理

私たちには話すべき多くのテーマがあります。あなたが使えそうなテクニックをたくさん目の前に投げかけていきます。これは基本的にデジタルコンピュータの性質、情報を処理するための汎用的な道具の性質によるものです。

コンピュータは何でもできる。私が「何でもできる」と定義すれば、それを証明することができるのです。つまり、数学的に証明できる、議論の余地のない、本物の証明だ。コンピュータは何でもできるわけではありません。これも証明できる。しかし、最も重要なことは、あなたと私だけがプログラムする限りにおいて、コンピュータはあまり多くのことはできないということです。これは経験則に基づく発見なのです。

So to offer guidance when the trade-offs become obscure, I am going to define the Basic Principle:

そこで、トレードオフが曖昧になったときの指針として、「基本原則(the Basic Principle)」を定義しておこうと思います。

- シンプルであること(Keep it Simple)

プログラムに追加する機能の数が増えれば増えるほど、プログラムの複雑さは指数関数的に増していきます。プログラムの内部的な一貫性はもちろんのこと、これらの機能間の互換性を維持する問題は、簡単に手に負えなくなります。基本原則を適用すれば、これを回避することができます。基本原則を無視したオペレーティングシステムをご存じかもしれません。

基本原則を適用することは非常に難しい。内外のあらゆる圧力が、あなたのプログラムに機能を追加するよう陰謀を企てます。結局のところ、それは半ダースの命令しか必要としないのですから、なぜそうしないのでしょうか? これら圧力に抗するのは基本原則のみです。基本原則を無視すれば、対抗する力はないのです。

基本原則を適用する助けとするために、あるルーチンでどれくらいの命令を使うべきかを説明します。また、ある機能を持ったプログラムはどれくらいの大きさであるべきか。これらの数値は、ほとんど機械に依存せず、基本的にタスクの複雑さを測るものです。これらの数値は、私が自分のプログラムで使用したルーチンを基にしているので、実証することができます。ここで警告しておきますが、私は4Kワードのコアに快適に収まるようなプログラムについて話すつもりです。

基本原則には一つの系(corollary)があります。

- 憶測は禁物です(Do Not Speculate!)。

使用するかもしれないというコードをプログラムに入れてはいけない。拡張をぶら下げるようなフックを置いてはいけません。あなたがやりたいと思うことは無限にあり、それぞれが実現する確率は0である。もし後で拡張機能が必要になったら、後でコーディングすればいいのです-そしておそらく、今やるよりも良い仕事ができるでしょう。そして、もし他の誰かがその拡張機能を追加したら、あなたが残したフックに気づくでしょうか? あなたはプログラムのその部分を文書化するでしょうか?

この基本原則には、もうひとつの系(corollary)帰結があります。

- 自分でやりなさい!です(Do It Yourself!)。

さて、いよいよ本題に入ります。これは、体制側との最初の衝突です。標準的なサブルーチンを使いなさい、というのが大なり小なり実施されている通例ですが、私は、「自分でサブルーチンを書きなさい」と言う。

自分でサブルーチンを書く前に、どう書くかを知らなければならない。 つまり、実質的には、以前に書いたことがあるということであり、そのため、なかなか着手することができない。しかし、試しに書いてみてください。同じサブルーチンを何台ものコンピュータや言語で何十回も書いているうちに、かなり上手になるはずです。そんなに長くプログラミングをするつもりがないのなら、この本に興味を持つことはないでしょう。

ご自身ではどのようなサブルーチンを書いているのですか? SQRTのサブルーチンに一目置いていました。トリッキーなもので、多くの才能を惹きつけるようです。が、ライブラリルーチンをうまく利用することができます。今やるなら入力サブルーチンです。彼らは岩の下から這い出てきたようです。FORMATステートメントが発明された15年前に作成されたワードが最新だいうのは、どうも納得がいかない。

後で詳しく述べますが、入力ルーチンはプログラムの中で最も重要なコードです。結局のところ、誰もあなたのプログラムを見ませんが、誰もがあなたの入力を見るのです。あなたの問題に全く興味を持たないシステムサブルーチンに身を委ねるのは、愚かなことです。出力サブルーチンやディスクアクセスサブルーチンにも同じことが言える。

さらに言えば、その作業は、あなたを躊躇させるほど大きなものではありません。汎用サブルーチンを書くには数百の命令が必要だが、必要なことは数十の命令でできる。むしろ、100命令より長いサブルーチンを書かないことをお勧めします。

もし、倍精度整数や複素整数を読み込みたければ、COBOLの入力サブルーチンに頼ったり、メーカーが改訂するまで待つことをしないでください。 自分で書いた方がずっと簡単だ。

しかし、もしみんなが自分自身のサブルーチンを書いたとしたら? それは、プログラムが機械に依存せず、全員が同じ言語で、もしかしたら同じコンピュータで書いているような千年王国から、一歩後退しているのではないだろうか? 私はこう考える。私には世の中の問題を解決することはできません。運が良ければ、私は良いプログラムを書くことができます。

## 1.2 プレビュー

プログラムの書き方を説明します。これは特定のプログラム、つまり特定の構造と能力を持ったプログラムです。特に、単純なものから複雑なものへの道筋、明確に定義された道筋に沿って拡張することができ、同様に単純なものから複雑なものまで、幅広い問題を扱うことができるプログラムである。このプログラムが考える問題のひとつは、まさに「複雑さ」の問題である。アプリケーションが必要とする以上にプログラムが複雑にならないように、どのようにプログラムをコントロールすればよいのだろうか?

まず、「入力」を定義し、入力があるかないかにかかわらず、すべてのプログラムに適用されるプログラミングの一般的な規則について述べます。実は、私たちはほとんど入力にしか関心がないので、入力を持たないプログラムについて言うべきことはあまりないのです。

入力を受け入れることで、プログラムはある制御言語を獲得し、それによってユーザはプログラムを可能性の迷路の中を導くことができるようになります。当然、プログラムの柔軟性は増しますが、それを正当化するためには、より複雑なアプリケーションが必要になります。しかし、実装の道具として制御言語が必要であることを認識することで、プログラムをかなり単純化することが可能である。

次の段階は問題指向言語である。プログラムが制御言語を動的に変更できるようにすることで、能力の質的変化を示すことができる。また、私たちの関心は、プログラムから、プログラムが実装する言語へと変わる。これは重要な転換であると同時に、危険な転換である。なぜなら、解決策の美しさの中で、問題を見失うことは簡単だからです。

ある意味で、私たちのプログラムは、アプリケーションに適用する言語を記述するメタ言語へと発展してきたのです。しかし、メタ言語と言いはしましたが、私は今後この言葉を二度と使わないようにします。その理由を説明したいと思います。物事はかなり複雑で、特に哲学的なレベルではそうなるのはお分かりでしょう。私たちの状況を正確に記述するためには、言語とメタ言語の2つのレベルでは足りない。少なくとも4つのレベルが必要なのです。 これらのレベルを区別するためには、微妙な議論が必要で、明瞭さではなく、混乱を促進するような議論になります。それに加えて、これらのレベルは実際にはしばしば入れ替わることがあり、そのため哲学的な議論が重箱の隅をほじくる議論になってしまう。

問題指向型言語は、私が遭遇したどんな問題も表現できる。そして忘れてはならないのは、私たちは言語ではなく、言語を動作させるプログラムに関心があるということです。言語を修正することで、同じプログラムを多くのアプリケーションに適用することができる。しかし、言語の拡張には、別の質的変化をもたらすものがある。それは、プログラムの能力を高めるのではなく、言語の能力を高めるのです。つまり、言語の表現力を高めるのです。第8章では、そのような拡張をいくつか考えてみます。私がこれらの拡張を集めたのは、主に、私がそれらの可能性をよく理解していないという共通の性質があるからです。例えば、英語の概念を応用した言語だと思う。

最後に、このプログラムを機械語に実装するためのプロセスを説明したい。つまり、基本的なプログラムが自分自身を修正し、拡張することができるブートストラップ技術である。

私が説明する考え方が、あなたにとって価値あるものであることを願っています。特に、私が説明するプログラムには、ある種の必然性があること、つまり、あることをしなければならない、ある順序でしなければならない、ある一定の規則が最適解を生み出すことに同意していただけると幸いです。

私は簡略化するためにある種の努力をしました。基本原則に反するようなことがないようにと願っています。プログラムを詳しく説明する方が、それを基本に落とし込むよりはるかに容易だからです。私の基本的なルーチンの上に自由に構築していただいて構いません。ただし、便利さを追加していることを認識してください。もしあなたが、便利なものと必要なものを混同するならば、あなたのプログラムは決して成長を止めることはないと私は保証します。

フローチャートがないことにお気づきでしょう。私はフローチャートが好きではない。なぜなら、フローチャートには無駄な情報を山のように含んでいると思えるからです(少なすぎたり多すぎたり)。また、プログラムの構造が通常よりも厳格になることを意味している。私は、あなたが何をすべきか、どのようにすべきかについて、かなり具体的に説明するつもりです。しかし、私は図ではなく、言葉を使うつもりです。仮に図を提供したとして、あなたが図に相応の注意を払うとは思えません。さもなくば、私が図を用意する際に相応の注意を払うでしょう。

# 2. 入力を持たないプログラム

最も単純なプログラムは、入力を持たないプログラムである。これはいささか愚かしい発言ですが、もしあなたが私に説明の機会を与えてくれるなら、いくつかの有用な定義を確立することができます。

まず「入力」という言葉について考えてみましょう。この言葉をある特定の意味で使いたい。

- 入力とは、プログラムを制御する情報である。

特に、私は以下を入力とは考えていない。

- コンピュータ内のメディア間でデータを移動させること。例えば
    - 例えば、テープをディスクにコピーしたり、ディスクをコアにコピーしたりすること。
- コンピュータにデータを読み込ませること。これはまさにメディア間の移動です。
    - カードからコアへ。

しかし、データには、データを識別したり、廃棄したりするための情報、つまり、入力が混在していることが非常に多い。例えば、80カラムのコードは、カードを識別するものです。これは入力であり、カードの残りの部分はデータであろう。

オペレーティングシステムはコントロールカードを使って、どのファイルを割り当てるか、どのサブルーチンを収集するか、などを指定します。このような情報は、間違いなくオペレーティングシステムへの入力です。あなたのプログラムの動作に影響を与えるかもしれませんが、あなたのプログラムがオペレーティングシステムそのものでない限り、あなたのコントロール下にはないので無視してください -

入力の認識をより鮮明にするために、入力を持つプログラムについて説明しましょう。計測したデータ点から滑らかな曲線を描くプログラムを考えてみましょう。このプログラムを実行するには、データ点の数、点間の間隔、実行する反復回数、そしておそらくどの関数を当てはめるかといった多くの情報が必要です。 この情報はプログラムに組み込まれている場合もあるが、そうでない場合は入力として与えなければならない。プログラム全体の目的である測定データそのものは入力ではない。理解できるようにするための入力を伴わなければならない。

入力のないプログラムは、非常に複雑である可能性がある。入力がないということは、言われなくてもプログラムが何をすべきかを知っているということでしかない。 コードに組み込まれているのは、実行に必要なすべての情報なのです。プログラムを再コンパイルしようと思えば、入力なしでプログラムを修正することさえできる。

しかし、私はプログラムを入力側から見ることにしています。私は、入力の複雑さによってプログラムをランク付けし、入力の複雑さを少し増やすだけで、プログラムの複雑さが大幅に減少することを実証するつもりです。 この観点からすると、入力のないプログラムは単純です。

私は入力について話すつもりなので、入力のないプログラムは何も話すことがありません。しかし、プログラム一般についていくつか指摘したいことがあるので、ここで述べておこう。ひとつは、私たちは木に登ることになります。高い枝に到達したとき、根を気にせずバランスを保つのに十分な苦労をすることでしょう。

## 2.1 言語の選択

我々は、多くのプログラマよりもコンピュータ言語にはあまり興味を持つべきではない。理由は3つある。第一に、我々はいずれ自分自身のアプリケーション指向の言語を定義することになる。その言語をどのように実装するかは、あまり関心がない。第二に、あなたはおそらく言語を選ぶ立場にはないでしょう。インストールしたことによって、選択肢がゼロになってしまっている。 第三に、我々は言語レベルの問題について話すつもりはない。

この最後のコメントについては、詳しく説明する必要があります。私は、あなたがすでに有能なプログラマであると仮定しています。私は、コンピュータがどのように動くのか、あるいは言語がどのようにコンピュータを隠しているのかを教えることに興味はないのです。私は、すべてのプログラムに共通する問題について、機械に依存せず、言語に依存しない方法で話をしたいのです。実装の詳細については、皆さんにお任せすることにします。私はプログラムを書くつもりはなく、プログラムの書き方を紹介するつもりです。

私は、あなたがコンピュータ用語で考えることができるほど優れたプログラマであることを望みます。つまり、誰かが自分のアプリケーションについて説明するとき、それをコンピュータの操作という観点から解釈するのです：ここではループ、ここでは計算、ここでは決定。細かいことはどうでもよくて、プログラムの構造が重要なのだ。

このデータはテープから取り出される、このループは……で停止する、これは実際には3方向の分岐である、といった具合に。特定のハードウェア構成に必要な問題として問題を修正するのです。

最後に、プログラムを特定の言語に翻訳する必要があります。 FORTRANではループを逆に回せない、COBOLでは3方向分岐がない、データにアクセスできない.....など、新たな問題に遭遇する。. . 現在の言語は、この最後のコーディング作業に必要以上の制約を課しているのです。

言語についてもう少し話したいのですが、ほとんど最も抽象的なレベル、つまりコンピュータ用語の話にとどまるでしょう。私たちはメタ言語だけで話をするわけではありません。私がインデックスレジスタをロードしろとか、負ならばジャンプしろとか言っても、それをあなたのコンピュータと言語に相当するものに翻訳してもらわなければなりません。

さて、高級言語が抱える大きな欠点を見てみよう。機械に依存せず、幅広い用途に適用できるようにしようとすると、コンピュータの能力のほんの一部しか利用できないことになる。コンピュータのループ制御命令の数と、言語のループ構成の数を比べてみれば、私の言っていることがわかると思います。

ここで、人気のある3つの言語について、その制限された能力を説明するために、1文で特徴を述べてみましょう。

- FORTRANは、複雑な代数式の評価に優れています。
- COBOLはパックされた10進数のデータを処理するのが得意です。
- ALGOLは、ループや条件文に優れている。

それぞれの言語は、その種の仕事には非常に効率的である。しかし、複雑な10進数表現を含む条件付きループが必要な場合は、問題があります。

私たちは、効率にこだわろうと思っています。効率的でなければ、まったくできないようなこともやってみようと思います。このようなことのほとんどは、高級言語の枠組みには収まりきらない。 また、コンパイラが許さないようなハードウェアの制御を必要とするものもあります。たとえば、FORTRANのサブルーチンを入力すると、使用するレジスタを保存することがあります。もしそれらを保存する必要がなければ、時間と空間を無駄にすることになります。ALGOLのサブルーチンは、あなたが使おうと予約したレジスタが使用可能であることを期待するかもしれません。その場合、あなたはそれらを保存しなければなりません。コンパイラとのインターフェースにかかる労力は、その見返りとして得られるものよりも多いかもしれません。

さらに、どの言語も物事を動かすのは得意ではありません。ほとんどの文はデータ転送です。あなたの最新のプログラムで数えてみてください。数字を移動させることでどれだけのことができるのか、そこには深い哲学的真理が隠されています。1つの命令でいくつものものを移動させたり、同じレジスタを何カ所にも配置したりできるのであれば、やらない手はないでしょう。

アセンブラでコーディングしなければならなくなります。どうしてもアセンブラで書くと言うなら、プログラム全体ではなく、これから集中的に学習する重要な部分を書いてください。FORTRANでもいくつかはできるかもしれませんが、努力に値するものではないというだけです。上位のサブルーチンがどこに行けるかをお見せしますが、その機能に制限する正当な理由があることに同意していただけると思います。

私はアセンブラの欠点は認識していますし、誰よりも欠点に悩まされています。でも、10倍ものカードに穴をあけてデバッグするのは好きじゃありません。でも、必要なパフォーマンスを得るため必要ならば、そうすることにしています。ところで、ここではアセンブラも含めて「コンパイラ」と呼ぶことにする。アセンブリ言語のプログラムをコンパイルするのだ。

後ほど、忘れ去られた言語である機械語でプログラムを書く方法を紹介しよう。つまり、コンソールの前に座り、スイッチで絶対的なバイナリ命令を入力するのだ。利用できるハードウェアとソフトウェア、そしてアプリケーションの性質によっては、機械語はすべての言語の中で最高の言語となるかもしれません。

## 2.2 コンピュータを選ぶ

もちろん、あなたがコンピュータを選べる立場にあるとは思っていません。また、ハードウェアの話をするつもりも全くありません。しかし、私はコンピュータの種類についての心象風景を持っており、それを説明することで、私の言うことのいくつかを理解してもらえるかもしれません。

例えば、16ビットワードで4kワード分のメモリと、一般的な命令セットと、必要であれば浮動小数点ハードウェアを持つ小さなコンピュータで、ほとんどのアプリケーションを非常にうまくプログラムすることができます。その場合、コンピュータはランダムアクセス可能な二次記憶装置(ここではディスクと呼ぶ)で補強される。ディスクの容量は重要ではなく、小さなディスクでも我々の目的には十分であり、アプリケーションによって決定される。 しかし、バックアップのために、ディスクを別のディスクやテープにコピーできることは重要である。そこで、2つの二次記憶装置と、入出力用のキーボードやカードリーダー、プリンタやスコープを備えた小型コンピュータを想定しています。

小さなコンピュータでアプリケーションを直列に実行する代わりに、大きなコンピュータで並列に実行することができます。1つのアプリケーションに使えるコアとディスクの量は、小型コンピュータで利用できる量とほぼ同じだからです。スピードは上がらないし、複雑なオペレーティングシステムが必要になるし、設備投資も膨大になります。しかし、私が考えているのは、4Kのコアと2次記憶装置、入出力装置という構成に変わりはありません。

## 2.3 配置とフォーマット

さて、ここからは言語やコンピュータに関係なく、プログラムの書き方を説明します。とうの昔ににやっていなければならないはずなのに、おそらく誰も教えてくれないのでやっていないこと。小さなことですが、もしあなたがそれをしなければ、良いプログラムを作ることはできません。

基本原則を忘れないでください もし「はじめに」を読んでいないなら、今すぐ読んでください。

すべての変数を宣言してください。FORTRANでさえもその必要がないというのになぜ? 誰もが、あなたが使っている、あるいは使う必要があると思われるパラメータを知りたがり、それを数えて、もっと少ないパラメータで済むかどうかを確かめたがるからです。あなたが言及せずに1つでも変数を入れてしまうと、いらいらする。

参照する前に、できる限りのことすべてを定義してください。FORTRANでさえその必要がないとというのに。なぜいけないのか? あなたもプログラムを逆に読むのは好きではないでしょう。「できる限りのことすべて」というのは、前方ジャンプを除くすべてという意味です。前方へのジャンプはあまりしないほうがいいでしょう。

変数はできるだけGLOBALにしてください。どうして? スペースが節約できるし、要件も明確になる。例えば、I、J、Kはいくつ必要ですか? ほとんどの場合、COMMONに1つコピーすれば十分でしょう(宣言しなければならないことを覚えておいてください、そして、COMMONに置いた方が良いでしょう)。もし必要なら、ローカルに再定義することができます、そして、あなたがしなければならないことは興味深いことです。

インデント! 高レベル言語や最近のアセンブラは、x列から開始することを主張しませんが、あなたはそうするのです。まっすぐ左マージンがそろっているという破格の魅力。紙は2次元です。使いこなせ。ループの中の文をすべてインデントすると、ループの範囲が一目瞭然になる。条件付きで実行される文をインデントすると、ネストした条件が自動的に整理されます。I = Iのような「入れなくてもいいのに」と思うような小さな文も、イン

デントすれば、リストを眺めていても邪魔にならない。常に同じ量のインデント、3スペース/レベルが良いでしょう。一貫性があり、正確であること。 インデントがずさんだと、一目瞭然です。

## 2.4 記憶しやすさ(Mnemonics)

読めばわかると思いますが、私はあるテーマについては強い意見を持ち、他のテーマについては全く意見を持ちません。実は、すべてのテーマについて強い意見を持っているのですが、時々、どの意見を表明するか決めかねてしまうことがあるのです。 幸いなことに、それはあなた自身に決断を委ねるものです。

記憶しやすいワードを使いましょう。残念ながら、あなたにとって記憶しやすいワードは、私にとっては記憶しやすいワードではないかもしれませんし、それを判断するのは私です。また、残念ながら、記憶しやすいワードは長くなりがちで、これと相反することになります。

短いワードを使いましょう。あなたは長いワードを打ちたくないでしょうし、私はそれを読みたくはありません。COBOLでは、これはダッシュを避けることと、修飾語を避けることを意味します。ダッシュと就職語は、状況によっては有益なこともありますが。

そこで、妥協案として、一貫した方法で省略し、自分自身のルールに固執することを提案します。私は、あなたが使っているルールを理解することができます。コメントで触れてもいいかもしれませんね。

文法的に正しい意味合いのワードを使うこと。変数には名詞を使う、サブルーチンには動詞を使う、.....には形容詞を。賢いワード(GO TO HELL)は使わないでください。そのかわいらしさはすぐに失われますし、その記憶的な価値は主観的すぎるからです。その上、あなたの性格まで見透かされてしまいます。

コメントは控えめに (これは歓迎すべきことでしょう。) あなたが目を通したプログラムを思い返してください。すべてのコメントに目を通しましたか? そのコメントはどれほど役に立ったでしょうか? どのくらいで読むのをやめましたか? プログラムは、たとえアセンブラのプログラムであっても、ニーモニックの助けを多少借りれば、セルフ・ドキュメント化されるものなのです。

- LA B . AにBをロードする

というのは意味がない。

このようなコメントを見ると、読むのをやめてしまい、役に立つコメントを見逃してしまうからです。

コメントで何を言わねばならないか、それは、プログラムが何をしているのかです。どうせ説明書からそれをどのように実行しているかを把握しなければならないのです。このようなコメントは歓迎です。

- COMMENT SEARCH FOR DAMAGED SHIPMENTS
  (破損した貨物の検索)

記憶しやすさは変数名とラベル名に適用されます(FORTRAN文番号でさえも記憶できるのでしょう?)。可能であれば、レジスタにもそれらを適用すべきです。同じ実体にいくつかの名前を付けて、現在の用途を示すとよいでしょう。しかし、名前を必要としないものに名前をつけて無駄な労力を使うのはやめましょう。カウンタが必要なら、I, J, K を使ってください。重要でない変数に大きな名前(EXC-CNTR)を付けても、何の役にも立たないでしょう。

## 2.5 ルーチンとサブルーチン

正確な定義を確立する必要がある術語が2つあります。サブルーチンは、来たところから戻る命令の集合体です。 ルーチンは、ある標準的な場所に戻る命令の集合です。

言い換えれば、ルーチンにジャンプして、サブルーチンを呼び出すということである。 この違いは、高級言語でも保持されています。GO TOとCALLやENTERの違いです。

だから何? サブルーチンはネスト(入れ子)に悩まされます。サブルーチンの中からサブルーチンを呼び出す場合、元のリターンアドレスを何らかの方法で保存する必要があります。これを実現するハードウェア / ソフトウェア的な方法はいくつも挙げることができますが、これらはすべて高くつきます。

戻ってくるつもりがなく、どこかにジャンプするのであれば、手間も時間もスペースも節約できる。ただし、本当に戻ってこない場合に限る。サブルーチンの呼び出しをシミュレートすることは、これまで以上に悪いことだ。

高水準言語は、自動的にネストすることでこれを隠している。最良の解決策は、ネストしなければならないときはするけれども、ネストしなければならないときだけすること、そして同じアドレスを2回以上保存しないことです。つまり、サブルーチンに入るとき、他のサブルーチンを呼び出すつもりなら、リターンアドレスを保存しておく。そして、最終的に戻る準備ができたら、ネストを解除する。

当たり前? そうかもしれない。でも、たいてい間違ったやり方でやっているのです。ハードウェアによっては、再帰的なサブルーチン呼び出しのときだけ問題が発生することもある。リエントラントプログラミングでは、常に問題が発生します。

こうして、ルーチンやサブルーチンの出入りが可能になりました。パラメータはどのように渡すのか? これもまた、コンピュータ、言語、プログラマの数だけ答えがある。標準的なものを挙げると、渡せるものはレジスタで渡し、残りはプッシュダウンスタックで渡す。

ルーチンが効率的に通信できることは非常に重要である。FORTRANのサブルーチンコールのコストについてご存じでしょうか。私はこれをこの言語の基本的な欠点と考えています。私たちは非常に多くのサブルーチンの間を移動するので、オーバーヘッドを最小化することに失敗すると、実行速度が簡単に半減してしまうかもしれません。

また、サブルーチンの価値も考えなければならない。サブルーチンは、論理関数を分離し、繰り返し命令を省くことができる。前者が受け入れ可能なのは、それが最小限のコストで済むときだけです。後者が受け入れ可能なのは空間が節約できるときだけです。1命令のサブルーチンは馬鹿げており、2命令のサブルーチンは3箇所から呼び出されて始めてとんとんです。気をつけましょう。

最後に、レジスタを効率的に使うことが重要です。レジスタを特定の目的のために割り当て、一貫して使用します。競合を避ける必要があれば、レジスタを再割り当てしてください。データをあるレジスタから別のレジスタに移動させないでください。そもそもそれがあるべき場所にあることを確認します。

私がレジスタというとき、アセンブラをイメージしていることは明らかです。しかし、他の言語でも添え字などを使ってレジスタの機能をシミュレートする必要があり、同じような配慮が必要です。

# 3. 入力のあるプログラム

入力のないプログラムは、単一のタスクを持つプログラムである。入力を持つプログラムは、多くのタスクを持つこともあり、それらは入力の指示によって実行される。したがって、私は入力を制御情報をみなし、制御情報を制御言語を定義するものとみなす。

この章では、ループを議論しているため、問題が発生することになる。 ループの各要素はその前任者と後任者に依存しており、私たちはどこから手をつけてよいのかわからない。私はできる限りのことをしましたが、物事を定義する前に参照することを余儀なくされています。特に次の章では、その直後に遭遇することになる細部のいくつかを示すことを正当化しようとします。

この章は、私が書き始めたときに予想した以上に、詳細でいっぱいです。 言わねばならないことの多さに私は驚くばかりですが、そのすべてが価値あるものだと思います。ただ、大事なのはプログラムの構造、コンセプトですから、細部に気を取られないようにしてください。

舞台を整えるために、このプログラムがどのように動作しなければならないか、簡単に説明しましょう。 あなたはキーボードの前に座り、入力を入力しています。あなたは文字列を入力し、コンピュータはそれをワードに分解します。コンピュータはワードを辞書で探し、辞書の項目で示されたコードを実行します(場合によっては項目で指定されたパラメータも使用します)。ワードを読み、それを識別し、そのワードに対するコードを実行するというプロセスは、確かに珍しいことではない。私は、このプロセスを体系化し、必然的な機能を抽出し、それが効率的に実行されることを確認しようとしているだけなのです。

## 3.1 名詞と動詞

辞書について述べましたが、この辞書を実装するために必要な詳細については、すぐに検討することにします。しかし、最初に個々のエントリについて少し話して、私たちが何をしているのかを感じてもらいたいと思います。

入力されたワードを読み、辞書で検索し、そのコードを実行するのです。ある種のワードはリテラルと呼ばれ、それ自身を識別するワードです。

- 1 17 -3 .5

このようなワードは辞書には載っていませんが、見た目で識別することができます。このようなワードは辞書にあるかのように動作し、実行されるコードはそれらをプッシュダウンスタックに配置します。

他のワードは、このスタックにある引数に対して動作します。

- ◦ スタックに置かれた最後の2つの数字を足し、その合計をスタックに残します。
- , スタックの一番上にある数字を印字し、スタックから削除する。
- 1 17 + ,

というようなフレーズを入力すると、これは、「1をスタックに置き、17をスタックに置き、それらを足して、その合計を入力する」という意味です。それぞれのワードは、他のどのワードとも関係なく、特定の限られた機能を果たします。しかし、ワードの組み合わせは、何か役に立つことを実現する。実際、次のように入力すると

- 4837 758 + -338 + 23 + 4457 + -8354 + ,

と入力すると、各数値が前の数値の合計に加算され、その結果がタイプされるのである。

これが今回のプログラムの価値である。単純な演算を柔軟に組み合わせて、タスクを達成することができるのです。

ここで、先ほど使ったワードをもう少し詳しく見てみよう。英語では2つのクラスがあり、それぞれに名前がついている。

- 名詞は引数をスタックに置く。
- 動詞はスタック上の引数を操作します。

すべてのワードは、コードを実行させる。ただし、名詞の場合、コードはほとんど実行されず、単にスタックに数字が置かれるだけです。動詞の場合は、その効果はかなり多様です。動詞は、2つの引数を追加する程度のものから、結果をタイプアウトする程度のものまであり、これには大量のコードが必要です。

事実上、名詞は、それらに作用する動詞を予期して、スタックに引数を配置します。予期(よき)という言葉は佳き言葉ですね。動詞をシンプルにするために、その引数が利用可能であることを約束するのです。次のワードを読んで、それを引数として使う動詞を定義することもできますが、一般的にはそうではありません。動詞はそれ自身の引数を提供することはありません。動詞を実行する前に、名詞を使って引数を提供します。実際、これによってプログラムが大幅に簡素化される。

エントリの特徴をもう少し拡張することができる。動詞は異なる数の引数を持つ。

- 単項動詞はスタック上の数値を変更する。
- 2項動詞は2つの引数を組み合わせて、1つの結果を残します。

算術演算は2項で、算術関数は通常単項である。しかし、動詞は分類しきれないほどたくさんあります。 例えば、スタックをタイプする動詞「,」は、スタックから数値を取り除くので、単項ではありません。しかし、この動詞は引数を1つ持っています。

動詞を区別するもう一つの方法は、以下の通りです。

- 破壊的動詞はスタックから引数を取り除きます。
- 非破壊動詞はスタックに引数を残す。

単項動詞と二項動詞、および型動詞「,」は破壊的です。 動詞DUPはスタックの先頭を複製するように定義しましたが、これは非破壊的です。一般に動詞は破壊的です。実際、私は破壊的な動詞とそうでない動詞を覚えるのを簡単にするために、意図的に破壊的な動詞を定義しています。あなたもそうすることをお勧めします。

リテラルは名詞です。他のワードも名詞として定義することができます。パラメータフィールドを使用してスタックに数値を配置するワードです。

- 定数は、そのパラメータフィールドの内容をスタックに配置します。
- 変数は、パラメータフィールドのアドレスをスタックに格納する。

例えば、PI が定数であれば、3.14 をスタックに格納します。このように

- 1. PI 2. * / ,

は、1.をスタックに、3.14 をスタックに、2.をスタックに、2.とPI を掛け算、1.を2PI で割り算、そして型付けと読み取れます。定数は、コード番号を使うときに特に便利です。そうしないと覚えられないような数字に名前をつけることができるからです。

しかし、最も重要な名詞は、リテラルと変数です。初歩的なプログラミングの教科書が苦労して説明しているように、変数は値ではなく場所に名前を与えるものです。しかし、高級言語では、変数には2つの使い方があることを隠しています。

- 値を取得する場所を指定する。
- 値を格納する場所を指定する。

定数は自動的に前者を実行し、後者を本質的に防ぎます(定数に値を格納することはできません)。 コンパイラのように文脈によって機能を区別するのではなく、変数に作用する2つの動詞を定義することにします。

- @ スタック上のアドレスをその内容で置き換える。
- = スタック上のアドレスに、スタック上のアドレスのすぐ下の値を入れる

したがって、Xを変数とすると、私が以下のようにタイプすると、

- X @ ,

つまり、Xのアドレスをスタックに置き、その値を取ってきて、タイプするのである。 そして、もし私が

- X @ Y @ + ,

とタイプすると、Xの値、Yの値を取ってきて、足して、印字する、という意味です。一方

- X @ Y =

は、Xのアドレスをフェッチし、その値をフェッチし、Yのアドレスをフェッチし、Xの値をYに格納する、という意味です。

- X Y =

では、Xのアドレス、Yのアドレスをフェッチして、XのアドレスをYに格納することだ。たぶん、これは私が言いたいことで、無理はないだろう。

先走った話になってしまうので、割愛します。しかし、変数には特別な動詞が必要で、そのうちの一つ(@)は通常明示されない。ちなみに、私はもともと@にVALUEというワードを使っていました。しかし、この動詞は頻繁に使われるので、一文字の名前にふさわしいし、@(アット)は、何か記憶的な価値があると思ったし、それ以外には何の役に立たないと思ったのだ。

ぜひ、動詞 @ を採用してください。いろいろな方法で隠すことができますが(後で説明します)、不必要に複雑になってしまいます。 これほど便利な動詞が使えないなんてありえない。その上、アドレスを変数に格納することができます。つまり、間接的アドレッシングです。

- X Y = Y @ @ ,

は、Xのアドレスを Y に格納する、Yのアドレスをスタックに置く、その値(Xのアドレス)をフェッチする、その値(Xの内容)をフェッチする、そしてタイプする、と読み取ります。

引数をスタックに載せて動詞で作用させる方法について、少しは理解していただけたでしょうか。定数と変数、単項と二項の動詞を定義しましたが、これらは単なる例に過ぎないことがおわかりいただけると思います。名詞や動詞、そしておそらく他の種類のワードも、あなたのアプリケーションに役立つように定義する必要があります。プログラミングとはそういうものだと思います。これから説明するようなプログラムがあれば、アプリケーションに必要な項目が決まれば、その項目をコーディングして問題を解決するのは至極簡単なことです。

## 3.2 制御ループ

私たちのプログラムは、見逃しがちな構造をしています。単一ループです。 しかし、このループは拡散しており、プログラム内のすべてのコードに散らばっています。そのため、ループの説明も必要です。

入力文字列からワードを読み、そのワードを辞書で調べ、そのワードが指定するルーチンにジャンプします。各ルーチンはループの先頭に戻り、別のワードを読みます。これから多くのルーチンを説明しますが、「別のワードを読むためにループの先頭に戻る」ことを識別するための用語があると便利です。ここではRETURN というワードを使いますが、同じ目的のために標準的なマクロやラベルをプログラム中に用意すべきです。

実際、あなたは2つの目的を達成します：ルーチンの終わりをマークします。 ルーチンの終わりを示すこと、そして、その前のコードがルーチンであること、サブルーチンとは異なるものであることを示すことです。このように、私は RETURN というワードを、FORTRAN の RETURN 文とは全く異なる意味で使っています。ここでは、サブルーチンからの EXIT について述べます。

制御ループには、パラメータスタックがその限界を超えていないかどうかのチェックが含まれていなければなりません。これは、ルーチンから RETURN した後に行うのが最適で、スタックを使用するルーチンに対してのみ行う必要があります。従って、2 つの可能な RETURN ポイントがあります(実際には 3 つ)。

制御ループは効率的でなければならない。もし、それが含む命令を数えるなら、あなたのプログラムに関連するオーバーヘッドを測定することになります。 あなたは、いくつかの非常に小さなルーチンを実行することになり、オーバーヘッドがマシンの使用率を占めているのを見つけるのは恥ずかしいことです。特に、パラメータスタック以外をチェックする必要はないでしょう。

もう1つのルーチンはこのセクションに属します：エラールーチンです。エラーが検出されるたびに、ルーチンは ERROR にジャンプして、問題のあるワードとエラーメッセージをタイプする必要があります。そして、全てのスタックと入力ポインタをリセットし、正常にRETURN します。

エラーメッセージをどのように扱うかという問題は重要な問題です。フラグの設定とテストを避け、サブルーチンの呼び出しでカスケードバックするのを避けるためです。リターンスタックをクリアすることで、保留中のサブルーチンのリターンを排除することができます。エラーフラグを付けて返さないことで、サブルーチンがエラーに悩まされることを避けることができます。これはコードを単純化するが、問題を処理する標準的な方法を持たなければならない。

キーボードに向かう人のイメージは、そのためにとても貴重なものです。 どんな問題が起こっても、「どうしよう」と悩む必要はない。 ユーザに聞いてみるのです。例えば、彼が辞書にないワードを入力したとします。どうすればいいのか? そのワードを入力すると、エラーメッセージ、この場合は「?」が表示されます。2つの数字を足そうとしたが、スタックには1つしかない。メモリの限界を超えたフィールドにアクセスしようとした場合、そのワードを入力して「LIMIT!」

もちろん、ユーザに解決できない問題を与えないように注意したいものです。MEMORY PARITY」というメッセージに直面したとき、彼はどうしたらいいのでしょうか? しかし、彼はプログラムよりも、ほとんどの問題に対して是正措置をとることができる立場にあることは確かです。そしてもちろん、どのような状況が問題であるかを決めるのは、あなた次第です。

ところで。ルーチンを実行した後までスタックをチェックしないので、いつの間にかスタックの限界を超えてしまいます。したがって、スタックのオーバーフローとアンダーフローは致命的でないはずです。良い解決策は、パラメータスタックをリターンスタックにオーバーフローさせ、メッセージバッファにアンダーフローさせることです。リターンスタックは決してアンダーフローしてはいけません。

## 3.3 ワードサブルーチン

プログラムを実行する制御ループを説明しました。このサブルーチンが最初に行うのはワードの読み取りです。 ということで、まず最初にワードの読み方について説明します。

ワードとは何でしょうか? コンピュータの言葉ではないことはお分かりだと思いますが、「ワード」という言葉をそのような意味で使わなければならないのです。ワードとは、スペースで区切られた文字列のことである。このルーチンは、より大きな文字列からワードを抽出するものである。

この定義をより一般的な入力ルーチンと対比してみよう。 例えばFORTRANのフォーマット入力では、ワードではなくフィールドが使われます。数値の意味はその数値が存在するフィールド、つまりカード上の位置によって決定される。私たちはカードを使わないので、位置という概念は不格好になり、その概念を使わずに順序に置き換えることになる。読むワードの順序は重要であるが、その位置は重要でない。 しかし、空のワードは認識できないので、フィールドを空にすることはできない。すべてのデータは明示的でなければならない。これはおそらく良い考えだが、習得には時間がかかる。オプションのパラメータを持つ入力規約は指定しないことにしよう。

では、WORDサブルーチンを書いてみましょう。入力ポインタはソーステキストの現在位置を指し、出力ポインタはワードを移動させるメモリ上の現在位置を指します。 ワードを移動させるのは、コンピュータのワード境界に合わせるためでもあり、ワードを修正するためでもあります。

入力文字を取得し、それが空白である限りは捨てます。 その後、別の空白文字を見つけるまで、それらを格納する(出力バッファにコピーする)。この空白文字を格納した後に、最後のコンピュータワードを埋めるのに必要な数のスペースをデポジットする。もしあなたが文字指向のマシンを持っているなら、私がワードアライメントにこだわる理由に興味を持つかもしれない。これは、検索サブルーチンで、ワード１つの大きさで比較することを想定しているためです。1ワードが6文字(あるいは2文字)であれば、たとえハードウェアがあったとしても、シリアルに比較するよりも並列に比較した方がはるかに効率的です。

ワードの長さに上限を設定するとよいでしょう。そのような上限は、使用する最大の数字を含むべきです。そこで問題になるのが、上限より長いワードをどうするかということです。そのワードを解剖する予定がないのであれば、余分な文字は単純に捨てることができるかもしれません(第8章)。それよりも、ワードの限界までスペースを入れる方がよいかもしれません。つまり、そのワードを2つのワードに分割するのです。おそらく、何か問題があって、その断片を処理しようとしているうちにその問題を発見するのだろう。しかし、この限界は、入力の実質的な制限にならない程度に、10〜20文字程度にする必要があります。また、コンピュータのワード長の倍数より1文字少なくすることで、整列したワードに必ず終端スペースが含まれるようにする必要があります。

ワードはスペースで区切られているのです。このような単純な定義には、おそらく異論があることだろう。例えば、算術式はしばしばワードとワードの間にスペースがありません。これについては、第9章で説明する。ここでは、潜在的な語彙を不当に制限しないようにするために、ピリオドやダッシュなどの文字をワードの中に埋め込む必要があるということだけを言っておこう。以下のこれらはワードであってほしい。

- 1,000 1.E-6 I.B.M. B&O 4'3" $4.95

## 3.3.1 メッセージ入出力

WORDサブルーチンは、おそらく入力文字を検査します。この文字はどこで手に入れるのでしょうか?

カードを読み込むことも可能ですが、ここではキーボードで入力することを想定します。さて、キーボードにはバッファドキーボードとアンバッファドキーボードの2種類があります。バッファドキーボードは、メッセージの終端文字を入力するまでメッセージを保存します。バッファリングされていないキーボードは、あなたが文字を入力するたびに文字を送信します。ハードウェアは、入力のバッファリングをするかしないかを選択できます。

いずれにせよ、私たちは各文字を複数回検査したいので、バッファリングされた入力が必要なのです。たとえ、入力された文字を受け取った時点に処理できるとしても、そうしない。メッセージバッファに格納しよう。

1行のメッセージバッファを確保する。その大きさは、入力でも出力でも、メッセージの最大サイズになりますから、132カラムのプリンタを使うつもりなら、十分な大きさにしてください。

バッファリングのシミュレーションを行う場合、バックスペース文字とキャンセルメッセージ文字を実装する必要があります。タイプミスを数多く行ってしまうからです。もし、あなたのハードウェアがバッファリングを行うが、これらの機能を提供しない場合は、バッファリングをシミュレートし、バックスペース文字とキャンセルメッセージもじゅを実装すべきです。これはおそらく入力のプリスキャンを意味するのでしょう。他の技法は複雑になりすぎ、おそらく最終的にはコストがかかります。

入力メッセージの終わりをend-of-message wordでマークする。これは他のワードと同じようにスペースで区切られたワードである。入力したメッセージの終端文字と一致するかどうかは、ハードウェアと文字セットによって、必要なスペースが確保できるかどうかが決まります。このワードは、メッセージの最後のワードをすぐに検出することができます。このワードは、特定の定義を持ち、貴重なタスクを実行します。

キーボードに加え、プリンタやスコープなど、何らかの出力デバイスが必要です。ここでもバッファリングされる場合とされない場合がある。入力と違って、バッファリングされていない出力を使わない理由はない。しかし、いくつかの出力デバイスがある場合、そのうちの1つはバッファリングされている可能性があります。もしそうなら、それらすべてをバッファドとして扱い、必要なところでバッファリングをシミュレートします。

入力と出力の両方に同じメッセージバッファを使うことにします。私の動機は、スペースの節約というか、スペースの利用率を上げることです。私の理由は、入力と出力は相互に排他的であるということです。 例外はありますが、通常、入力を読み、出力を準備することはないでしょう。少なくとも、そうする必要はない。

しかし、メッセージ・バッファがまだ入力を含んでいるかどうかを示すスイッチ(1ビット)は必要です。最初に(あるいは毎回)出力をタイプするときは、このスイッチをリセットしなければなりません。このスイッチは後で使用します。

入力メッセージが完全に揃ったときに終了する受信サブルーチンが必要です。同様に、出力メッセージを送信した後に終了する送信サブルーチンが必要です。もし、ハードウェアが確認応答を提供するのであれば、それを待つ必要があります。あるメッセージの送信と次のメッセージの準備を重ねて行うことはしないでください。送信は非常に遅く、準備は非常に速いので、目立った速度向上は望めません。 また、プログラムがかなり複雑になります。

### 3.3.2 文字を移動させる

文字の取り込みと格納については、主に入力と出力に関連して、何度かお話しします。例えば、WORDサブルーチンはメッセージバッファからワードバッファに文字を移動させます。概念的には簡単な作業ですが、実装するのは難しいものです。 配列を移動させるのも全く同じ問題です。しかし、実際には、配列を移動する必要はなく、文字を移動しなければならないのです。

入力ポインタと出力ポインタの2つの実体を定義してみましょう。後で一般化することになりますが、今のところインデックスレジスタと考えることができます。また、2つのサブルーチンを書きますが、ハードウエアによっては命令とすることも可能です。FETCH は入力ポインタで特定される文字をレジスタにロードし、入力ポインタを前進させ、DEPOSIT はそのレジスタを出力ポインタで特定される位置に格納し、出力ポインタを前進させます。

FETCH と DEPOSIT は、コンピュータによって非常に単純なものから非常に複雑なものまであります。もし、1命令以上を必要とするならば、サブルーチンにしておくとよいでしょう。これらを組み合わせることで、移動が可能になります。ただし、文字を格納する前に検査できることが重要です。ハードウェアの移動命令にはほとんど意味がありません。

入出力ポインタはインデックスレジスタを使用します。しかし、これらのレジスタは移動の間だけ使用されるべきです。なぜなら、それらは多くの目的のために使用され、そこに永久に何かを保存することは非現実的だからです。

# 3.4 10進数変換

入力文字列からワードを分離して整列させた後、制御ループはそのワードを辞書で検索します。もし辞書になければ、それは数字かもしれない。数値は、辞書に登録する必要のない特殊なワードです。ワードそのものを調べることで、そのワードをどう扱うかを決めることができます。数値に対して実行されるコードは、その数値のバイナリ表現をスタックに配置します。

スタックについては、次のセクションで説明します。まず、数をより正確に定義してみましょう。

## 3.4.1 数値

何が数で何が数でないかを正確に述べるのは非常に難しい。数字を2進数に変換するNUMBERサブルーチンを書かなければならないが、このサブルーチンが数字の定義である。このサブルーチンがあるワードを数字に変換できるとき、そのワードを数字とみなす。変換できなければ数字ではない。

ワードを調べてそれが数字であるかどうかを確認し、次にその数字を2進数に変換するのは愚かなことである。検査と変換は、非常に簡単に一つの処理にまとめることができる。

ワードの中には、必ず数字になるものがある。先頭にマイナスを付けることができる数字の羅列である。このような数値は通常2進数の整数に変換される。たとえば

- 1 4096 -3 7777 0 00100 10000000 6AF2 -B

は10進数、8進数、16進数の一部です。この数字は基底を特定せず、16進数かもしれないワードが10進数であるとは限りません。

つまり、すでに基数は複雑な数を持っているのだ。そして、単純な整数の先には、固定小数点数、浮動小数点数倍精度整数、複素数分数など、無限の種類の数字がある。そして、このような数は、小数点、暗黙の小数点、指数、接尾辞など、ワードとして様々な形式を持つことができる。実際、同じワードでも文脈によって異なる数を表すことがある。

どのような種類の数値が必要なのか、どのような書式にするのか、どのように変換するのかを決めることが、大きな仕事のひとつになります。各数値の種類はNUMBERサブルーチンによって一意に識別可能でなければならず、それぞれのために出力変換ルーチンを提供しなければなりません。

私は以下のガイドラインを提案します。常に整数と負の整数を定義すること、先頭のプラス記号を許容しないこと、数値としては意味がないがワードとしては使えること、浮動小数点ハードウェアがあれば浮動小数点分数を小数点で区別すること、浮動小数点ハードウェアがあるならば、固定小数点分数を小数点の有無で区別すること、浮動小数点シミュレーションをしないこと、分数に指数をつけないこと、分数上の指数を許可しないこと、などです。これらの規則により、NUMBERサブルーチンが単純になりますが、その概要を説明します。

あなたのアプリケーションでは、特別な数値フォーマットが必要かもしれません。

- 45'6 for 45 ft. 6 in., an integer
- 1,000,000 an integer
- $45.69 an integer

このような数字をNUMBERに含めることは難しくないが、可能な限りの形式を含めることはできない。中には互換性のないものもあります。

- 3'9 for 3 ft. 9 in.
- 12'30 for 12 min. 30 sec. of arc
- 12'30 for 12 min. 30 sec. of time
- 4'6 for 4 shillings 6 pence

基本原則忘れんなや!

固定小数点数はほとんど使われていません。しかし、私はその価値を確信していますので、皆様にそれをお示ししたいと思います。浮動小数点ハードウェアを使用する場合、固定小数点数はより大きな意味を持つという利点しかなく、それはおそらくあまり価値がないでしょう。しかし、浮動小数点ハードウェアがなければ、浮動小数点ソフトウェアの非常に大きなコストなしに、浮動小数点数の能力のほとんどを提供します。例外は指数(数値の桁数)の幅の広さです。

私は、指数がコンピュータ上でひどく誤用されていると確信しています。ほとんどのアプリケーションでは、机上計算機で使えるような実数、例えば10^6から10^-6の間の数を使います。このような数値は、固定小数点フォーマットでも同じように表現できます。浮動小数点は必要ありませんが、もしハードウェアが使えるなら使った方がよいでしょう。特に物理学の分野では、10^43や10^-13といった大きな指数が発生する場合がある。しかし、これは通常、適切な単位が選ばれていないか、あるいは対数を使うべきかを示している。

もちろん、コンパイラは固定小数点を実装していないので、人々は固定小数点を使いません。私たちはそれを実装し、固定小数点(整数)命令で可能となる速度を利用する立場にある。固定小数点数とはどのようなものでしょうか? 使いたい小数点以下の桁数を選びます。小数点以下の桁数は時々変更してもかまいませんが、異なる精度の数値を混在させてはいけません。NUMBERサブルーチンでは、すべての数値(小数点付き)を、その小数点以下の桁数を正確に入力したかのように整列させます。それ以後はその数値を整数のように扱います。つまり、小数点以下3桁を選択した場合、

- 1.は1.000とみなされ、1000として扱われます。
- 3.14は3.140で3140となります。
- 2.71828は、2.718で2718です。
- -.5は-.500で-500です。

アプリケーションで要求されるか、ハードウェアで簡単にできるのでなければ、わざわざ四捨五入する必要はないでしょう。

加算、減算は小数点以下が揃っているので、安心して行えます。乗算は、2つの数値を掛け合わせた後、1000で割って小数点を再び揃える必要があります。通常、ハードウェアがそれをやってくれます。乗算の結果は倍精度の積となり、被除数(devident)として適切な位置となります。2つの数を割る前に、精度を維持し、小数点を揃えるために、被除数を1000倍する必要があります。これも簡単です。

このように、必要な小数点以下の桁数を格納するのに十分な大きさのワードを用意すれば、固定小数点演算は簡単に行えます。ハードウェアがあれば、倍精度の数値と演算で、同じくらい簡単に、より大きな数値を扱うことができます。浮動小数点演算をシミュレートするよりもずっと簡単です。平方根や三角関数のサブルーチンを自分で書かなければならないかもしれませんが、利用可能な近似関数があるので、これは難しいことではありません。また、同等の浮動小数点演算シミュレートのサブルーチンよりもはるかに高速になります。

10進小数点の位置合わせは視覚的にわかりやすく、切り捨ての問題も回避できます。しかし、2進小数点を揃える方が良い場合もあります。つまり、小数点以下 3 桁ではなく、2 進小数点以下 10 桁で整列させるのです。そうすると、1000の乗算と除算はバイナリシフト

(2進数に相当)に置き換えることができ、より高速になります。変換(入力と出力)時のアライメントの問題や、乗除算時の切り捨てが微妙になることと、速度の利得のバランスを取る必要があります。また、演算の説明も難しくなります。

### 3.4.2 入力変換

さて、NUMBERサブルーチンの詳細について説明します。まず、なぜサブルーチンなのでしょうか? これまでのプログラム、そして巻末の増補版プログラムを見ても、NUMBERは制御ループの中で一度だけ呼び出されていることがわかると思います。私のルールでは、NUMBERはインラインコードであるべきなのです。しかしながら、NUMBERのロジックは非常に複雑です。サブルーチンの目的の一つである論理的機能の強調に基づき、制御ループから切り離して制御ループ自体の混乱を少なくしたいのです。また、他のルーチンからNUMBERを呼び出したくならないとは決して思いませんし、実際、私はそうしました。しかし、私は、このようなプログラミング標準の違反は、明確に認識しておくべきだと思います。

優れたNUMBERサブルーチンの鍵は、そのサブルーチンが呼び出すもう1つのサブルーチンにあります。このサブルーチンは2つのエントリポイントを持ちます。SIGNEDは次の文字がマイナスかどうかをテストし、スイッチを設定し、これまでの数をゼロにしてNATURALに落ちます。NATURALは文字を取得し、それが数字であるかどうかをテストし、これまでの数字を10倍し、その数字を加算します。これは数字でないものを見つけるまで繰り返されます。

このルーチンで、NUMBERは次のように動作します： 入力ポインタを整列されたワードの最初にセットし、SIGNEDを呼びます。もし停止文字が小数点の場合、カウンタをクリアし、NATURALを呼んで分数を取得し、カウンタを用いて10の累乗を一つ選択して浮動小数点または固定小数点に変換します。いずれにせよ、SIGNEDのスイッチを適用して、number-so-farを負にします。終了(Exit)。

NUMBERを呼び出すルーチンは、停止文字をテストすることができます。

- もし、それがスペースであれば、変換は成功した。
- そうでなければ、そのワードは数でなかった。

例えば、以下は数字である。

- 0 3.14 -17 -.5

以下は数字ではありません。

- 0- 3.14. +17 -.5Z X 6.-3 1.E3

いずれの場合も、NUMBER はスペース以外の場所で停止する。これまでの数値は、その時点(0 かもしれない)までは正しく変換されますが、それは何の価値もありません。

SIGNED/NATURAL は 2 回呼び出されるので、サブルーチンとして正当です。 さらに、他の数値書式を定義すれば、便利なことがわかります。例えば、書式 ft'in について、

- SIGNED を呼び出した後、もし停止文字が'であれば、これまでの数に 12 を掛け、NATURAL を呼び出します。その後、通常通り、小数点の有無をテストしながら進みます。

"in"(インチ部分)が 12 未満であることを確認したい場合は、これを少し修正する必要があります。

NATURAL では、これまでの数値に 10 が掛けられます。リテラルの 10 を使用するのではなく、フィールド(BASE)を定義し、そこに乗数として 10 を格納します。 それから BASE を 8 (または 16) に変更し、8 進数を処理することができます。 BASE を 2 に変更し、2 進数を使用することもできます。NATURAL は数字を BASE と比較してテストしなければなりませんので、8 進数で 9 は禁止されます。16 進数入力の数字では、標準の文字セットでは 9 の後に A-Z が続かないため、さらに問題が生じます。しかし、この問題は一箇所(NATURAL)に限定されており、コーディングは容易です。

- 原点は通常、2 進数の値を得るために数字から差し引かなければなりません。BASE が 16 の場合、A-F から別の原点が引かれます。

NUMBER は、少なくとも数字ではないワードを認識することにおいては、効率的であるべきです。数字をたくさん使うからというわけではなく、数字でないワードをたくさん調べるからです。これについては、第 8 章でさらに詳しく説明します。また、ワードのアライメントされたコピーを調べることも重要です。理由はいくつかあって、入力ポインタのトラブルを避けるため、終端スペースを保証するため、などです。 しかし、これには問題があります。使用する最大の数値は、アラインされたワードに収まらなければなりませんので、他の方法で使用するよりも長いワードが必要になる場合があります。ワードサイズより長い数値は、その右端の桁が破棄されます。これは、おそらくその数値の外観を破壊しないので、エラーは検出されませんが、変換は不正確になります。この問題は深刻ではありませんが、注意してください。

### 3.4.3 出力変換

数値出力は数値入力より難しいです、なぜなら余分なステップがあるからです。入力時には、数字を10倍して、各桁を足す。左から右へ作業することができます。出力では、10で割って、その余りを桁として保存し、商が0になるまで繰り返す。右から左へ数字が得られるが、左から右へ印字したい。

したがって、数字を一時的に保存する場所が必要です。メッセージ・バッファの一番奥が良い場所です。このスペースは、おそらく番号のために十分なスペースを持っているので、未使用です。もちろん、スタックを利用することもできます。一時保管場所の右端にスペースを置き、右から左へ数字を格納してゆけば、最終的にTYPEBサブルーチンを使って数字を打ち込むことができます。

負の数と分数の両方を扱いたいと思うことでしょう。 数が負であることを記憶しておいて、絶対値に対して作業します。 終了後、マイナスを前置します。分数は2つの変換ループが必要です。1つは分数を変換して桁数を数え、小数点を格納するもの、もう1つは整数を変換して商が0になったら止めるものです。分数の商をテストしたくないのです。

もしあなたが注意深く、2、3の命令を費やすなら、あなたは以下の方法で数字の外観を改善することができます。

- 小数点以下の数字がない場合は、小数点を入力しない。
- 小数点の左側にゼロを入力しない。

おそらく、異なる出力形式を指定する複数の辞書項目があることでしょう。例えば、整数、浮動小数点、複素数のようなそれぞれの種類の数値は、それ自身の出力ルーチンを必要とする。しかし、実際の変換は、特殊なケースを区別するためのパラメータを持つ単一のサブルーチンが行うべきである。すなわち、NUMBERサブルーチンの逆を行う単一のサブルーチンである。異なる数値間の類似性は、それらの間の差異よりもはるかに大きいのです。

10進固定小数点分数を使用する場合、すでに小数点以下の桁数を指定するフィールドDがあります。同じフィールドを使用して、出力上の小数の配置を制御します。通常、入力と出力の小数点以下の桁数は同じになります。浮動小数点数であっても、完全な精度の出力に興味を持つことはほとんどないため、このフィールドは必要です。

もしレポートを作成するのであれば、つまり、慎重にフォーマットされた数値の列を作成するのであれば、数値を右寄せにする必要があります。つまり、小数点以下を揃えるのです。このためには、もう1つのパラメータFが必要です。これは、数値が右寄せになるフィールドの幅です。使い方は簡単で、数値を右から左に変換した後、必要なスペースの数を計算し、SPACEを呼び出します。その後、TYPEBを呼び出す。スペースを決定する際、

TYPEBは常に数値の後にスペース1つを印字することを覚えておいてください。したがって、数字と数字の間には少なくとも1つのスペースが必ず入ることになります。もし、数字が指定したフィールドに収まらない場合、レポートの書式は乱れますが、スペースが一つ残るので、完全な数字が入力されます。

数字を右寄せにする場合、右から左へ直接数字を配置することができることは認めます。しかし、右端がどこになるかを知っておく必要があります。その場合、出力を開始する前にメッセージバッファをスペースで埋める必要があり、バッファのない出力ですぐに印字することはできません。しかし、私の最大の不満は、フリーフォーマット出力の組み立てができないことです。例えば、文中に数字を入れる場合、余分な先頭のスペースを入れないようにします。そして、非常に多くの場合、アンフォーマット出力で十分であり、気にしないフィールドサイズを指定する手間を省くことができます。

フォーマット要件に応じて、必要になる辞書項目が他にもあります。SPACE項目は、スタック上に指定した数だけスペースを確保します。スタックが負の値であれば、出力ポインタを変更することで後方に移動させることもできます。これは、TYPEBで提供されるスペースを抑制したい場合に便利です。タブエントリは、スタック上の特定の位置に到達するために必要なスペース量を計算 するかもしれません。

# 3.5 スタック

私たちはいくつかのプッシュダウンスタックを使うので、あなたがそれらを実装できることを確認したいと思います。プッシュダウンスタックは、後入れ先出しの方式で動作します。これは、配列とポインタで構成されています。ポインタは、配列に配置された最後のワードを識別する。あるワードをスタックに載せるには、ポインタを進め、そのワードを格納しなければならない(処理はこの順序で行う)。 スタックからワードを取り出すには、そのワードを取り出し、ポインタを下ろします(処理はこの順序で行う)。実際に押し下げることはないが、効果は同じである。

スタックポインタは、インデックスレジスタが十分な数があれば、インデックスレジスタの優れた利用法です。特にメモリへの加算命令がある場合、間接アドレッシングも可能です。

## 3.5.1 リターンスタック

このスタックは、戻り情報を格納します。用途としては、サブルーチン呼び出しでインデックスレジスタを使用する場合に、サブルーチンのリターンアドレスを格納することが挙げられます。 スタックの後入れ先出しの性質は、まさにネストされたサブルーチン呼び出しに必要な動作です。この後、同じスタックに格納できる他のいくつかの種類の戻り情報

に遭遇することになります。重要なことは、リターンスタックとパラメータスタックを一緒にしないことです。これらは同期していません。リターンスタックの容量は、おそらく8ワードで十分です。

### 3.5.2 パラメータスタック

このスタックは、私が単にスタックと言ったときに意図しているものである。後述するように、数値、定数、変数はすべてこのスタックに置かれる。このスタックは、ルーチン間でパラメータを渡すために使用されます。各ルーチンは、他にどれだけのパラメータがあるか、どれだけの時間前にそこに置かれたかに関係なく、そこに自分の引数を見つけることができます。パラメータスタックの長さが 16 ワード未満であるような実装をすべきではありません。

パラメータスタックに対する重要な改良点は、スタックの一番上にあるワードを保持するためにレジスタを確保することです。もし、この方法が問題を起こさないためには、いくつかのルールに従わなければなりません。

- このレジスタを他の目的に使ってはならない。
- このレジスタを常に満杯にしておかなければならない。空であることを示すフラグはありません。

もし、これらの条件を満たすことができないなら、スタックを完全にコアに置く方がよいでしょう。

いくつかの用語が必要です。

- スタックにワードを置くと、スタックのサイズが大きくなります。
- ワードをスタックから取り除くと、スタックのサイズが小さくなる。
- スタックの一番上にあるワードをトップワードと呼ぶ。
- スタックの一番上にあるワードのすぐ下のワードを下のワードと呼ぶ。

入力からパラメータスタックを制御する必要がある場合があります。これらのワード(辞書項目)は非常に便利で、上記の用語を説明するものです。

- DROP スタックから先頭のワードを削除します。
- DUP トップワードをスタックに入れ、それによって複製する。
- SWAP トップと下のワードを交換する。
- OVER 下のワードをスタックに入れ、トップワードの上に移動させる。

# 3.6 辞書

入力のあるプログラムはすべて、辞書を持たなければならない。入力のない多くのプログラムには辞書がある。しかし、これらは辞書として認識されていないことが多い。一般的な「カジュアル」な辞書は、IF ... ... のシリーズである。 ELSE IF ... ... ELSE IF ... 文、またはそれと同等のものである。実際、これは、辞書が小さく(8項目)、拡張不可能であれば、妥当な実装である。

辞書の機能と存在を認め、辞書を一箇所に集中させ、項目の形式を標準化することが重要である。悪いプログラムに共通する特徴は、辞書に相当するものがプログラム中に散在し、空間的、時間的、そして見かけ上の複雑さにおいて大きな犠牲を払っていることである。

エントリの最も重要な特性は、通常見落とされるものです。各エントリは、実行されるルーチンを特定する必要があります。 多くのエントリが同じルーチンを実行することはよくあります。おそらく、選ぶべきルーチンがほとんどないのだろう。このため、各エントリで何を実行するかを指定することの重要性が隠れてしまいがちです。各エントリにルーチンのアドレスを配置することで、そのコードに到達するための最適かつ標準的な手順を設計することができる。

重要なのは、IF ... . ELSE IF 構文には、各エントリにルーチンを関連付けるという特徴があります。

## 3.6.1 エントリの形式

辞書項目を形成する方法には、2つの方法があります。ハードウェアの特性によって選択することができるが、私は2番目の方法を推奨する。エントリの主な特徴は、可変長であることである。 エントリの一部は、実行されるコードであったり、パラメータであったり、記憶領域であったりするが、これらはすべて任意の長さを持つことができる。

1つのエントリを固定サイズと可変サイズの2つの部分に分割することも可能です。これにより、固定サイズのエントリをスキャンしてワードを特定することができ、多くの場合、この検索を高速化するハードウェア命令が存在します。固定サイズのエントリの中に可変長エリアへのリンクを置くこともできます。もちろん、例外としてオーバーフローするようなリンクになるように、固定サイズを選択します。

しかし、入力は比較的少量であるため(定義を増やしても)、辞書の検索に要する時間を最小にすることは、全体最適にはつながらない。可変長の項目を直接連結することで、より柔軟に、よりシンプルにコアを割り当て、最終的にはより高速にすることができる。これが、これから述べる構成である。

エントリには、定義されるワード、実行されるコード、次のエントリへのリンク、パラメータの4つのフィールドがあります。それぞれについて説明する。

ワードの形式はワード入力ルーチンとともに決定されなければならない。ワードのサイズは固定で、NEXT で定義されたサイズより小さくてもよいが、ハードウェアのワードサイズの倍数でなければならない。 しかし、より洗練されたアプリケーションは、出力メッセージを構築するために辞書のワードを使用する。その場合、ワードを切り詰めないことが重要であり、その場合、ワードフィールドは可変長でなければならない。このフィールドのサイズをマークするために、文字カウントではなく、空白文字を使用する必要がある。可変エントリ内で可変ワードフィールドを扱うには、ワードは一方向に(後方に)、パラメータは他方向に(前方に)伸びる必要があります。固定または可変ワードサイズのどちらを選ぶかは、基本原則の適用が必要です。

コードフィールドには、テーブルや他の省略形へのインデックスではなく、ルーチンのアドレスを入れるべきです。プログラムの効率は、3.9で説明するように、エントリが特定された後、コードにたどり着くまでの時間に強く依存します。しかし、プログラムのサイズが小さいと、このアドレスはハードウェアアドレスフィールドより少ないスペースに収まることができます。

リンクフィールドも同様に、ハードウェアで指定されたものより小さくてもよい。これは、現在のエントリからの距離ではなく、次のエントリの絶対位置を含むべきである。

パラメータフィールドは、通常4種類の情報を含む。

- 定数または変数で、サイズは可変です。数値の性質は、実行されるコードによって決定されます。
- 配列 - 数値が格納されるスペースです。配列のサイズはパラメータであるか、または実行されるコードに含まれているかもしれません。
- 定義：仮想のコンピュータ命令を表す辞書の項目の配列。3.9を参照してください。
- 機械語命令: プログラムによってコンパイルされたコードで、このエントリが実行される際にこのコードが実行される。このようなデータは、おそらくワード境界でアラインされなければならないが、他はその必要がない。

### 3.6.2 検索戦略

基本的原則が、辞書の検索に対して適用されます。これは、最新から最古の項目へと逆方向でなければならないということです。辞書は、エントリが行われた順番以外(例えば、アルファベット順など)には配列されていないことにお気づきのことと思います。これによって、同じワードが再定義された場合に、最新の意味が得られるのです。この性質を損なうほどの価値のあるトレードオフはありません。

ワードを特定するには、そのワード(またはその最初の部分)をレジスタに入れ、各エントリ(またはその最初の部分)と等しいかどうかを比較する。代数的な比較で十分である。ワードを浮動小数点数として扱うと、誤った等式が成立するのではないかという懸念があるようです。 この可能性はゼロだし、ワードはいつでも変更可能なので無視しましょう。

高速化のため、(文字単位ではなく)ワード全体での比較を行うべきである。通常、マッチは最初の部分で見つかり、拡張部分はあまり効率よく扱われないかもしれません (それでも全ワードの比較は可能です)。

固定長のエントリは、単純なループでスキャンすることができる。リンクされたエントリは同様に単純なループを必要とするが、ループ実行はより遅くなる。 しかし、リンク検索の速度は無制限に向上させることができる。 各項目をその物理的に前方に存在するエントリにリンクするのではなく、いくつかの連鎖のうちの１つの前方エントリにリンクする。ワードを入力するときにも、検索するときにも、どの連鎖に属するかを判断するためにスクランブルをかける。このように、ワードを見つけるため、あるいはワードがないことを保証するために検索する必要があるのは、辞書全体のほんの一部だけです。

鎖の数は2のべき乗であるべきで、8であれば有用な速度向上が得られる。スクランブル技術は、最初の数文字を足し合わせて低次ビットを使用するという、非常に単純なものでよい。連結辞書を維持するためには、次に利用可能な場所と、最後のエントリの場所を保持しなければならない。多重連結辞書は、各連鎖の最後のエントリの位置を必要とする。これは、大きな時間増加のための小さなスペースの代償である。

しかし、検索時間は重要な考慮事項ではないので、辞書が非常に大きい(数百の項目)場合を除き、多段鎖を避けることを勧める。

### 3.6.3 初期化

辞書はプログラムに組み込まれており、おそらくコンパイラによって初期化されます。これは、固定サイズの項目がある場合に特に当てはまります。 しかし、可変サイズの項目は互いにリンクされなければならず、特に複数の連鎖を持つ場合、これはコンパイラの能力を超えている可能性がある。

このような場合、辞書をスキャンしてリンクを確立するループを書くのは簡単なことである。辞書が占めるコアをスキャンし、何らかのユニークなフラグ(リンクフィールドの7)によってエントリを認識する必要があります。このループはワードを拾ってスクランブルし、適切なチェーンに追加することができる。

これは純粋に一時的なコードである。スクランブルとリンクのための永久的なサブルーチンを呼び出すかもしれませんが、初期化コードはそれ以上使用することはありません。したがって、プログラムの進行に伴ってオーバーレイできる場所に配置する必要があります。十分な大きさがあれば、メッセージバッファやディスクバッファに配置する可能性もあります。

その他のもの、特に特定のタスクが割り当てられているレジスタの初期化が必要かもしれません。このような任務はすべてこの一箇所に集中させるべきです。

## 3.7 制御言語とその例

アプリケーションは、面白くなる前に複雑になりがちです。 しかし、ここに制御言語の利点を示す、かなり一般的な問題があります。実装は厄介で、実行は非効率的だが、プログラムは単純で、そのアプリケーションは柔軟だ。

ここでの問題は、シーケンシャルなファイルを調べて、あるレコードを選択し、ソートして、いろいろな方法でリストアップすることである。レコードのフィールドを以下の変数で定義するとする。

- NAME AGE SALARY DEPT JOB SENIORITY
  名前 年齢 給料 部署 仕事 年功序列

これらの動詞を定義してみよう。

- LIST SORT EQUAL GREATER LESS

各動詞は、次の例のように、その前に置かれた動詞が生成した一時ファイルに作用する。

部門6に属するすべての従業員をアルファベット順にリストアップする。

- 6 DEPT EQUAL NAME SORT LIST

まず、dept = 6 のレコードを選び、一時ファイルにコピーします。そして、そのファイルを名前でソートする。そして、それをリストアップします。

dept 3 で 17 の仕事を持つ従業員を年功序列で 2 回リストアップする。

- 17 JOB EQUAL 3 DEPT EQUAL SENIORITY SORT LIST LIST

給与が 10,000 ドル以上の従業員を年齢別にリストアップし、年功序列が 3 未満の従業員を特定する。

- 10000 SALARY GREATER AGE SORT LIST 3 SENIORITY LESS LIST

いくつかのコメントが示されているようです。論理的な "and"はいくつかの select 動詞を順番に使用することで適用できますが、論理的な "or"は使用することができません。 不必要にレコードを並べ替えないのであれば、複数のフィールドでソートすることができます。さらに 2 つの動詞が必要である。

- REWIND END

は、元のファイルからやり直したり、終了したりするためのものである。

実際には、特定のレコードを探し出して修正する機能など、他にも多くの機能を提供することができます。しかし、特定のアプリケーションを設計するのではなく、名詞と動詞を組み合わせることで、簡単なプログラムでも大きな柔軟性が得られることを示したいのです。 このような簡単な例でも、ワードサブルーチン、数字サブルーチン、辞書、スタックなど、私たちのすべての機能を使用していることに注目してください。 これは推測ではなく、本質的なコードを提供しているのです。

# 4. 成長するプログラム

これまでのところ、私たちの辞書は静的なものです。必要な項目はすべて含まれており、プログラムがコンパイルされたときに配置されたものです。しかし、そうでなければならないということはありません。 項目を定義することで、さらに項目を作成したり削除したりすることができます。なぜ、このようなことが望ましいのか、説明しましょう。

あるアプリケーションを制御するプログラムがあるとします。あなたが入力したワードをもとに、あなたが指示したとおりに動作します。第3章では、結果をタイプアウトする機能を提供しました。アプリケーションの必然的な結果ではなく、あなたが見たいと思うような変数を出力するのです。 入力によって直接制御されるので、より会話的な出力になります。

この状況には、2つの問題があります。まず、辞書に項目を追加するためには、プログラムを再コンパイルしなければなりません。明らかに、多くの項目を追加することはないでしょうが、もしかしたら、その必要はないかもしれません。第二に、すべての項目が同時に存在しなければならない。これは、容量の問題というより、複雑さの問題です。アプリケーションが複雑な場合、すべての局面で互換性を持たせることはますます難しくなります。例えば、すべてのフィールドに明確な名前を見つけることです。第三に、入力に誤りを発見した場合、プログラムを再コンパイルしなければなりません。もちろん、この能力をもつように項目を定義することは可能であるが、そのような能力はない。

もし、辞書の項目を作成することができれば、2つのことを達成することができる。アプリケーションの異なる別の側面にプログラムを適用することができます - 衝突することなく、複雑さを軽減することができます。辞書エントリを異なる方法で作成することができ、エラーを修正することができる。実際、あなたのプログラムの目的は、徐々に変化してゆくが、この変化は重要なものです。 あなたは、アプリケーションを制御するプログラムから始めました。今、あなたはアプリケーションを制御する機能を提供するプログラムを持っています。 実質的に、言語からメタ言語へとレベルアップしているのです。 これは非常に重要なステップです。生産的でないかもしれません。これは、アプリケーションと話すことから、アプリケーションについて話すことへとあなたを導きます。

このレベルアップを別の面から見れば、辞書の項目にもそれが表れています。最初は、アプリケーションプログラムを構成するコードの断片を実行するためのワードでした。純粋な制御機能です。今は、アプリケーションプログラムを構築するためのワードになってい

ます。問題志向の言語である。この違いは、突然生じる必要はないが、不可逆的である。あなたは、アプリケーションからシステムプログラマになり、システムがあなたのアプリケーションとなるのです。

これが良いのか悪いのか、判断に迷うところです。もうお分かりだと思いますが、これはアプリケーション次第なのです。十分な複雑さを持つアプリケーション、そしてきっと一般性を持つアプリケーションならどのようなものも、特殊な言語を開発しなければならないのではないでしょうか。制御言語ではなく、記述言語です。

いくつかの例を挙げましょう。シミュレータは制御言語を必要としない。重要なのは、シミュレート対象となるシステムを非常に効率的に記述できることです。線形計画問題は、その問題を記述できる言語が必要である。コンパイラは、実際にコンパイルするプログラムに対して記述言語を提供します。コンパイラ・コンパイラはコンパイラを記述するものです。記述したコンパイラを実行し、コンパイルしたプログラムを実行できるコンパイルコンパイラとは何だろうか? それが問題なのです。

ここで、あなたが記述言語を適用するにふさわしい問題を持っていると仮定します。どのような辞書項目が必要でしょうか?

# 4.1 辞書項目の追加

辞書を拡張したい、つまり、専門言語を正当化できるほど十分に複雑なアプリケーションを持っていると仮定してみよう。どのように辞書エントリを作成するのでしょうか?

制御ループについて考えます。制御ループは、ワードを読み取り、辞書を検索します。あるワードを定義したい場合、制御ループにそのワードを見せないようにしなければなりません。 その代わりに、次のワードを読み、それを使ってから制御ループに RETURN するエントリを定義しなければなりません。事実上、それは次のワードを見えなくする。ループはワードサブルーチンを呼び出す必要がある。このサブルーチンは、ルーチンではなく、サブルーチンであるのはそのためです。このようなエントリを定義エントリと呼ぶことにしましょう、その目的は次のワードを定義することです。

原則として、定義するエントリは1つだけでよいのですが、そのエントリが定義するコードのアドレスをパラメータとして与えなければなりません。各エントリには、ワード、コードアドレス、リンク、パラメータ(オプション)の4つのフィールドが必要であることを思い出してください。ワードサブルーチンから取得するワード、構築するリンク、スタックから取得するパラメータです。アドレスはスタックから取得することもできますが、構築

するエントリの種類ごとに別の定義語を用意した方が便利です。つまり、必要なアドレスごとに別の定義項目を用意し、その定義項目のパラメータ・フィールドからアドレスを取得するのです。

しかし、これでは混乱するかも知れませんね。新しいエントリのアドレス・フィールドをパラメータ・フィールドから提供するエントリが1つあります。例えば、ある定数を定義するとします。

- 0 CONSTANT ZERO

これは、0をスタックに置きます。CONSTANTのコードは、次のワードZEROを読み込んで辞書エントリを構築します。前のエントリへのリンクを確立し、スタックからパラメータフィールドに0を格納し、自身のパラメータフィールドからZEROが実行するコードのアドレスを格納します。これはおそらく、パラメータフィールドの内容をスタックに格納するコードのアドレスです。

このように、これから作成するエントリの種類ごとに、コードのアドレスを提供し、作業を行うための定義エントリが必要です。すべての定義エントリは共通なので、それらのエントリが呼び出せる ENTRY サブルーチンを作成する必要があります。このサブルーチンはコードアドレスをパラメータとして持ち、パラメータフィールドを除くすべての新しいエントリを構築しなければなりません。

他の定義項目は以下のようなもの「かもしれません」。

- `0 INTEGER I` - 整数サイズのパラメータフィールドは0に初期化され、そのアドレスはスタックに配置されます。
- `1. REAL X` - 浮動小数点パラメータ・フィールドは1に初期化されます。
- `8 ARRAY TEMP` - 8ワードのパラメータ・フィールドが0にクリアされ、その1ワード目のア ドレスがスタックに置かれます。

私は「かもしれない」という言葉を強調しておきます。アプリケーションが異なれば、定義エントリも異なるでしょうし、同じワードでもアプリケーションが異なれば動作も異なるかもしれません。しかし、あなたは、必要なあらゆる種類の名詞を定義し、その名詞のインスタンスを好きなだけ作成できる立場にあるのです。普遍的な名詞のセットを確立しようとするのは無駄な作業である。コンパイラ言語では、十分な数の名詞を用意しないことで何度もつまずき、いくら用意しても、誰かがもう一つ欲しくなる。

たとえば、次のような名詞を定義することができる。

- `0 8 INDEX J` - Jは、0から8までのインデックスと定義される。実行されると、その値がスタックの一番上に追加される。

そして、Jを進めたり、テストしたり、リセットしたりする適切な動詞を定義すれば、強力なインデックス機能を手に入れることができる。あるいは、次のように記述することで、

- 3 VECTOR X 3 VECTOR Y 9 VECTOR Z

ベクトルZを定義し、以下のようにベクトル演算を実現する演算動詞を定義する。

- `X Z = Z Y +` X と Y を加算し、Z に格納する。
- `X Y Z *C` XとYを乗算(外積)しZに格納します。

アプリケーションに必要なものは何でも定義することができます。しかし、すべてを定義することはできない。基本原則の出番です。

## 4.2 エントリを削除する

これまで、名詞の定義についてだけ説明してきました。実は、名詞よりも動詞のほうが使う数が多いのですが、動詞にはもっと長い説明が必要です。 ここでは、動詞の一種を紹介します。

辞書に項目を追加することができれば、いずれは項目を削除したくなるものです。正しく入力し直すために項目を削除したり、別のアプリケーションのために項目を削除したりする必要があります。結局のところ、辞書は有限であり、どんなに大きくしても、その上限を意識することになる。パーキンソンの法則は、こう言い換えることができる。辞書は、利用可能なスペースを埋めるために拡張する。

項目を削除する方法は一つしかない。それは、ある時点以降のすべての項目を削除することです。特定の項目を削除すると、辞書は連続したコアを占有するため、穴があいてしまう。 その穴埋めをしようとすると、絶対アドレスを使っているため、リロケーションの問題に直面する。絶対アドレスを避けることは、非効率的であり、不必要である。

末尾のエントリを削除することは、完全に満足のいく解決策である。このことを証明する議論は、試してみる以外にはありません。実際には、多くの項目を追加し、問題を見つけ、それらの項目を削除し、問題を修正し、すべての項目を再入力することに気づくだろう。あるいは、あるアプリケーションで辞書を満たし、それを消去し、別のアプリケーション

で再度満たし、といった具合です。あるいは、いくつかのフィールドをクリアするために、同じアプリケーションを再ロードすることもあるでしょう。いずれの場合も、最新の項目をすべて削除したい。

例外は、いくつかのエントリを使用して他のエントリを作成する場合です。構築したエントリはもはや必要なく、それらを取り除く方法はありません。このようなことはよくあることで、後でいくつか例を挙げるかもしれない。 しかし、失うのはたかだか辞書のスペースだけで、実用的な解決策は見当たりません。

では、末尾の項目を削除するにはどうしたらよいのでしょうか。辞書のある一点をマークして、その位置にすべてのものをリセットしたいのです。一つは、辞書の中で次に利用可能なワードを識別する辞書ポインタです。これは簡単です。しかし、検索チェーンのそれぞれについて、前のエントリを識別するチェーン・ヘッドをリセットする必要があります。これは小さなループで済みます。検索するときと同じように、指定したポイントの前にあるリンクを見つけるまで、それぞれのチェーンをたどって戻ってください。

固定サイズのエントリがある場合、パラメータ領域へのポインタをリセットする必要がありますが、リンクをたどる必要はありません。

削除したいポイントを指定する便利な方法は、そこに特別なエントリを置くことです。実行すると、それ自身とそれに続くものを削除する動詞です。例えば

- REMEMBER HERE
  (ここを覚えておいてください )

は定義エントリです。HERE と入力すると、それは忘れ去られます。それは、辞書の場所をマークし、削除コードを実行します。HERE はパラメータ・フィールドを必要としませんが、固定長のエントリを使用する場合は、パラメータ・ポインタの現在値を保存する必要があります。 これが動詞を定義するエントリの最初の例です。

# 4.3 操作

スタックは引数を見つける場所であることを思い出してください。演算機能を提供するために定義したいワードがいくつかあります。これらは制御言語にとってはほとんど価値がありませんが、パワーを付加するためには不可欠です。TRUE (1)とFALSE (0)という論理構文を使うことにします。そして、3.6のtopとlowerの定義を思い出してください。

単項演算子：スタックの一番上の数値を変更します。

- MINUS は top の符号を変更します。

- ABS は符号を正にします。
- ZERO は、top が 0 の場合、それを TRUE で置き換えます。さもなくば、FALSE をスタックに置きます。
- NONZERO は、top が 0 でない場合、TRUE をスタックに配置します。さもなくば、それをそのままにしておきます(FALSE をスタックに残します)。

二項演算子：top をスタックから取り除き、lower を両者の関数で置き換えます。

- + lower に top を足す。
- * lower に top を掛ける。
- - lower から top を引く。
- / lower を top で割って、商を残す。
- MOD lower を top で割って、余りを残す。
- MAX top が lower より大きい場合、lower を top で置き換えます。
- MIN top が lower より小さい場合、lower を top で置き換えます。
- ** lower を top 乗する。

これらはサンプルに過ぎない。これらは単なるサンプルであり、有用と思われるワードを自由に定義してよい。引数を操作する前にスタックに置かなければならないことに注意してください。数値は自動的にスタックに置かれます。定数も同様です。したがって、次のようにすればよい。

- 1 2 +
- PI 2. *
- 1 2 + 3 * 7 MOD 4 MAX
- 1 2 3 + *

この表記法は、電卓と同じような感覚で算術計算ができる。よく括弧のない表現とか、右辺のポーランド語とか言われるが、これは単にスタック上の引数を扱う方法である。従来の代数的記法は実装がかなり難しくなる(8.2 参照)。

他の二項演算は算術関係である：これらはスタック上に真理値を残す。

- = それらは等しいか?
- < top は lower より大きいか?
- > top は lower より小さいか?
- >= は top が lower より大きくないか?

- <= top は lower より小さくないか?
- 1 2 +
- PI 2. *
- 1 2 + 3 * 7 MOD 4 MAX
- 1 2 3 + *

論理演算には単項演算子と二項演算子がある。

- NOT top が FALSE の場合は TRUE に、それ以外の場合は FALSE に置き換えます。
- OR 論理和演算
- AND 論理積演算
- IMP 論理包含(lower が偽または top が真のときに真となる)
- XOR 論理的排他的論理和

スタックは固定ワード長でなければなりません。しかし、上記の演算は、整数、固定小数点数、浮動小数点数、倍精度分数、複素数、上記の種類のベクトルなど、いくつかの種類の数に適用されるかもしれません。真理値は1ビットだけである。明らかに、スタックはあなたが使用すると予想される最大の数を保持することができなければなりません。しかし、様々な種類の数をどのように区別するかは、あまり明確ではありません。

1 つの方法は、それぞれの種類の数に対して別々の演算を定義することです。

- ◦ 整数と固定小数点の加算(これらは同じです)。
- +F 浮動小数点の加算
- +D 倍精度の加算。

もう 1 つは、スタックエントリを十分に長くして、数値の種類を識別するコードを含むようにすることです。この場合、各操作を定義するコードがより精巧になり、不正な引数をどう扱うかの問題が発生します。

私は、単純化するために、引数をチェックせず、別々の演算を定義することをお勧めします。実際、一度に一種類の数しか扱わないので、問題は発生しないかもしれません。

混合モード算術をわざわざ使う必要はない。決して必要ではないし、面倒の大きさに値するほど便利でもない。複数ワード(複素数、倍精度)の場合、数値のアドレスをスタックに置くことがあります。しかし、これでは3アドレス演算になってしまい、一般に結果が引数の1つに置き換わってしまう。 そしてこれは、ひいては定数に関する複雑な問題を引き起こすことになる。

一般に、数字を使ってできることは無限に増えていく。その多くは互いに相容れないものです。基本原則を思い出せ。

# 4.4 定義エントリ

ここで、これまでのどの作品よりも複雑なエントリについて説明しなければならないが、見ていただくものの中で最も複雑な作品というわけでもない。また、オプションでない点も例外的です。あるワードを他のワードで定義できるということです。ワードはそれ自体では単純だが、組み合わせると強力になると説明したのを覚えているだろうか。では、ワードを組み合わせる方法を紹介しましょう。

定義は、「:」という定義項目と、「;」で終わる一連のワードで構成されます。これは、「:」で定義されたワードが、その後に続くワードによって表現される意味を持つことを意図している。たとえば、以下のようになります。

- : ABS DUP 0 LESS IF MINUS THEN ;

これはABSというワードを定義したものです。その目的は、スタック上の数値の絶対値を取ることである。これは、適切な効果を持つ一連のワードを実行することによって行われます。

これはABSの定義としてはかなり不器用なものだと思うかもしれません。特に、あなたのコンピュータには、まさにそれを実行する命令があるのですから。おっしゃるとおり、定義は不器用になりがちです。しかし、そのおかげで、私たちは先見の明なく定義しておかなかったワードを使うことができるのです。ある基本的なワードがあれば、どんな項目でも作ることができるのです。定義によって、制御言語と応用言語が簡潔に区別される。制御言語にはすべての機能が組み込まれていなければならず、アプリケーション言語には必要な機能を構築することができるのです。

定義の実装は簡単ですが、厄介なことに微妙です。定義のパラメータフィールドには、それを定義する辞書エントリのアドレスが含まれています。これらの項目を何らかの方法でパラメータ領域に格納し、後で定義を実行するときにそれらを取得する必要があります。定義と実行の相補的なプロセスは、これまで出会ったどの項目よりも複雑です。

これらの処理を詳しく説明する前に、定義とは何かということを正確に明らかにしておこうと思う。ワードに対して実行されるコードはルーチンであり、サブルーチンではないことを思い出してください。しかし、一連のワードは、制御ループが次のワードを見つけることができる位置に戻る機能を果たすので、一連のサブルーチンの呼び出しのようなものです。定義とは、まさにサブルーチンのアドレスが定義を構成する一連のサブルーチン呼び出しであると考えることができるかもしれません。

もうひとつの視点は、私が使っている略語に隠されている。私が「ワードを実行する」と言うとき、本当はそのワードに関連するコードを実行することを意味します。より正確には、そのワードの辞書に登録されているアドレスのコードを実行することである。この略語は便利なだけでなく、ワードが実行可能な命令であることを示唆しています。実際、ワードを命令と考えることは有用である。すなわち、我々の現実のコンピュータがシミュレーションしているコンピュータに対する命令である。この架空のコンピュータを「バーチャルコンピュータ」と呼ぶことにしよう。つまり、ワードを入力するということは、仮想のコンピュータに指示を出していることになる。制御ループは、仮想コンピュータの命令フェッチ回路になる。

これを定義に拡張すると、定義はバーチャル・コンピュータのサブルーチンになる。そして、定義を行う作業は、このサブルーチンをコンパイルすることに相当する。このアナロジーには後で再度触れる。

バーチャル・コンピュータは、定義を理解する上で本当に役に立つことがおわかりいただけると思います。実際、私はこの仮想計算機のおかげで、コンパイラの技術を定義に応用するようになりました - 他の方法では思いつかなかった技術です。 しかし、プログラマにとっては便利でも、プログラマでない人には混乱を招くだけです。そこで私は、この種の項目には「定義」という名前をつけ、その説明として「あるワードを他のワードで定義する」という言葉を好んで使っています。

定義は非常に強力です。なぜ協力かを説明することは難しいし、理解するのさえ難しいです。その価値は、後知恵で理解するのが一番だ。あるアプリケーションの実装が非常にシンプルになったとき、12個の定義を使い、それらが8階層にネストしていることに気がつきます。 定義があるからこそ、シンプルになったように見える。

しかし、サブルーチンの呼び出しと比較して、定義の価値を強調するいくつかの性質があります。第一に、呼び出しの順序、どのレジスタが利用可能でどのレジスタを保存しなければならないかなどを気にする必要がなく、ただワードをタイプするだけでよい。第二に、ある定義が別の定義を実行することができる。つまり、定義を入れ子にすることができます。この場合も、戻り値の保存や他のレジスタとの競合を気にする必要はありません。また、再帰的に定義を使用することも可能です。第三に、引数はスタック上にあるので、定

義間で簡単に、実際には見えないように渡すことができます。ここでも呼び出し順序やストレージの競合を心配する必要はありません。一時記憶領域も十分にあり、これもスタック上にあります。

FORTRANのサブルーチン呼び出しに比べれば少ないかもしれないが、この便利さの代償はもちろん存在します。その代償が制御ループです。これは純粋なオーバーヘッドである。各エントリのコード実行はもちろんコンピュータのスピードで進みますが、次に実行するコードのアドレスを得るのに、8命令ほどかかります。このため、私は制御ループを最適化することを強くお勧めします。

もし、ワードのために実行されるコードが制御ループに比べて長ければ、そのコストは無視できるほど小さくなることに注意してください。これは制御言語の原理です。コードが制御ループのサイズにまで小さくなると、オーバーヘッドは50%以上に上昇する。これがアプリケーション言語の代償である。ただし、アプリケーションプログラムをサポートするOSやコンパイラを使えば、50%のオーバーヘッドには簡単に達してしまうことに注意してください。

妥協することをお勧めします。計算に限られる部分はコード化し、それ以外の部分は定義を使ってください。計算を実行するのではなく、制御するために定義を使用することは安く済みます。 また、定義の作成が簡単なため、実装にかかる時間と労力、ひいてはコストを削減することができます。

### 4.4.1 定義を定義する

定義エントリ `:` は、他の項目と同じように動作する。これは、アドレス `EXECUTE` を `ENTRY` サブルーチンに渡します。このコードについては、次のセクションで説明します。

そして、スイッチ `STATE` をセットします。 `STATE` が0であれば、すでに説明したようにワードが実行され、1であればワードがコンパイルされる、というように制御ループを変更する必要があります。繰り返しになりますが、プログラムに定義を追加する場合は、制御ループを変更して、ワードの実行とコンパイルのどちらかを行うようにしなければなりません。もし、最初から定義を入れるつもりなら、それに合わせて制御ループを計画しなければなりません。ワードの実行ができるだけ速くなるように、スイッチを実装してください。コンパイルするワードの数よりも、実行する数の方がはるかに多くなるはずです。

ワードをコンパイルすることは単純です。辞書でワードを見つけたら、その辞書エントリのアドレスを入手する。このアドレスをパラメータ・フィールドに格納する。私たちはすでに、辞書にワードを登録するためのメカニズムを持っています。ENTRYは、多くの定義エントリと同様に、パラメータに対してこれを使用しています。辞書ポインタDPは、辞

書内の次に利用可能なワードを指す。ワードをコンパイルするためにしなければならないことは、DPにそのアドレスを格納してDPを進めることだけです。また、実行されたコードのアドレスではなく、*エントリ*のアドレスを格納することに注意してください。これは、コードだけでなく、パラメータ・フィールドや、必要であればワード自体にもアクセスできるようにするためです。

さて、ワードをまとめるのはここまで。では、数値はどうだろうか。コンパイラに提示される数値はリテラルと呼ばれます。そして、リテラルはどのコンパイラにとっても問題です。幸いなことに、バーチャルコンピュータがインラインでリテラルを扱えるように定義することができます。数値が正常に変換されたときにSTATEをテストするように、再び制御ループを変更する必要があります。

数値をコンパイルする方法を示す前に、擬似エントリを定義しておきます。疑似エントリは、辞書に載っていない辞書の項目です。つまり、エントリの形式を持つが、他のエントリにリンクされていない。したがって、辞書検索で見つかることはない。 仮想コンピュータをスムーズに動作させるためにエントリが必要になることがありますが、参照できない項目を含めることで辞書検索の速度を落としたくはないですよね。

お察しの通り、リテラルをコンパイルするためには、擬似エントリをコンパイルします。そして、その後に数字そのもの、つまり、数字もコンパイルします。その結果、倍長の仮想計算機命令ができあがります。擬似エントリのために実行されるコードは、数値をフェッチしてスタックに配置する必要があります。このように、コンパイルされたリテラルは、実行されたときに、即時(immediately)実行されたのと同じ効果があります。

もし、異なるサイズのリテラルがあれば、それらに異なる疑似エントリが必要であることに注意してください。そして、その話題が出たところで、ワードの長さについて少し議論しましょう。仮想計算機のワード長は12ビット程度とすべきです。これは、各命令が単に辞書のアドレスで構成されており、12ビットあれば、おそらく1000のエントリのうちの1つを識別するのに十分だからです。実コンピュータのワード長が18ビットより長い場合は、いくつかの仮想コンピュータの命令を1ワードにまとめる必要があります。DPを実計算機のワード以外を指すアドレスに変更しなければならないので、厄介なことになるかもしれません。しかし、多くのスペースを節約することができます。

ちなみに、リテラルはコンパイル時に余分なスペースを必要とするので、よく使うリテラルをワードとして定義するとよいでしょう。

- 1 CONSTANT 1

数値の変換が行われる前に辞書が検索されるため、数値はワードになる可能性があることを思い出してください。また、ワードは1つの長さの仮想計算機命令を必要とするだけです。一方、辞書の項目はコンパイルされたリテラルよりもはるかに多くのスペースを必要とするので、そのトレードオフに注意してください。

制御ループの中でワードをコンパイルするコードでは、";"に注意する必要があります。 これは通常通りコンパイルされますが、それ以上コンパイルが進まないようにSTATEをリセットします。また、別のタスクも実行されますが、これには少し説明が必要です。

定義をコンパイルしているとき、それぞれのワードについて辞書を検索していることに注意してください。もし、今定義したワードを参照すれば、それを見つけてしまいます。つまり、再帰的な参照をしていることになります。再帰的な定義が必要なら、それはそれで構わない。しかし、再帰性を再定義に置き換える方が絶対に便利である。つまり、定義の中の自分自身への参照が、それ以前の定義を参照していると理解することだ。 たとえば

- : = SWAP = ;

ここでは、引数を逆の順序で操作するために、=動詞を再定義しています。この目的のために別のワードを使うこともできるが、=にはニーモニックとして意味がある。

いずれにせよ、この機能を提供するのは簡単である。最新のエントリが見つからないように、「:」で検索にバグを出します。そして、「;」で検索を解除して、新しい定義を有効にします。もし再帰的な定義が必要なら、「:R」という定義項目を用意すれば、「;」が両方に対して機能するようにすれば、バグを出さないようにすることができます。もう一つのテクニックは後で述べます。

### 4.4.2 定義を実行する

定義のために実行されるコードをEXECUTEと名付けました。これは仮想計算機の命令フェッチ回路を変更しなければならない。

制御ループの構造を思い出してください。ルーチンNEXTWが辞書項目のアドレスを提供し、この項目に関連するルーチンが入力され、最終的にNEXTWに戻る。NEXTWがNEXTIに置き換わることを除いて、定義を実行するためには同じ手順が必要である。NEXTWがワードを読み込んで辞書で見つけたのに対し、NEXTIは定義のパラメータフィールドから次のエントリをフェッチするだけである。

したがって、次のエントリのために入るルーチンを識別する変数が必要である。一つの実装は、NEXTWまたはNEXTIのアドレスのいずれかを含むフィールドNEXTを定義することである。もしNEXTに間接的にジャンプすれば、適切なルーチンに入ることになる。したがって、EXECUTEの１つのタスクは、NEXTIのアドレスをNEXTに格納し、後続のエントリを別の方法で取得させることです。

もちろん、NEXTIは次のエントリを見つける場所を知っていなければなりません。ここで、仮想コンピュータのアナロジーは、命令カウンタを追加することによって拡張されます。ICというフィールド(できればインデックスレジスタ)を定義すれば、実際のコンピュータの命令カウンタとまったく同じように動作させることができます。これは、次に実行されるエントリを特定するもので、実行中に進まなければなりません。

IC で特定されるエントリをフェッチし、IC を次のエントリに進め、NEXTW と同じポイントに戻ってエントリを実行する(場合によってはコンパイルする)、これが NEXTI の完全な動作です。定義を使用する場合は、広範囲に使用することになります。ですから、NEXTIはNEXTWの代償として最適化されるべきです。特に、エントリを実行(コンパイル)するコードは、NEXTIから落ち、NEXTWからジャンプするようにします。これにより、NEXTIを使った制御ループの命令(ジャンプ)が１つ節約できる。これは、エントリのコードを実際に実行するのとは別に、ループの20%を占めることができ、大幅な節約になります。

ここで、EXECUTEに戻ります。NEXTIの確立に加えて、ICを初期化する必要があるのは明らかです。しかし、その前にICを保存しなければならない。このプロセスは、仮想コンピュータのサブルーチン呼び出しに似ています。IC を保存する場所は、明らかにリターンスタックです。これは他の目的にも使われますが、いずれもそのような使い方と矛盾するものではありません。ある定義が他の定義から実行される場合、現在のICが保存されなければならないことは明らかです。そうでなければ、ICの現在の値は未定義である。

この処理には、もう一つのルーチンが関与しています。";"を実行したコードは、定義から戻らなければなりません。つまり、リターンスタックからICをリストアしなければならない。しかし、EXECUTE で NEXTI に設定された NEXT の値も復元しなければなりません。リターンスタックに古いNEXTの値を保存して、";"にそれを復元させることができます。もっと簡単なのは、IC の未定義値をゼロにして、NEXT を NEXTW に戻すフラグとして機能させることでしょう。定義実行中は、NEXTには常にNEXTIが含まれます。ソーステキストに由来する定義から戻るときだけ、NEXTWが再確立されなければならない。ソーステキストの実行中はICは関係ないので、このような限定的な方法で役に立つかもしれません。

それだけである。EXECUTE、NEXTI、";"の組み合わせは、強力で効率的なサブルーチン機能を提供します。定義に対して「実行」されるコードは、先に議論したように、フィールドSTATEによって実際にコンパイルされるかもしれないことに注意してください。また、定義によって実行される項目は、他の項目をコンパイルするかもしれないことに注意してください。つまり、ある項目がDPを使用して辞書に数字を格納するかもしれない。このように、フィールドICとDPは、DPはエントリを格納し、ICはエントリを取り出すという類似した使い方をしていますが、両者は同時に使用されることがあります。インデックスレジスタが不足している場合でも、まとめようとしないことです。

### 4.4.3 条件

定義のプロセスを簡単におさらいしておこう。ワード「:」は、制御ループを変更するスイッチを設定します。このスイッチは、ワードを実行するのではなく、コンパイルするようになります。ワード「;」はコンパイルされるが、同時にスイッチをリセットし、コンパイルのプロセスを終了させる。次のワードは通常通り実行される。

というのも、「;」はある意味例外的なワードで、コンパイル中に実行され、その際にスイッチがリセットされるからです。もちろん、定義の実行中にも実行され、ICをリセットするという別の効果があります。

他にも「；」のように、コンパイル中に実行されなければならないワードがあります。これらのワードはコンパイルを制御する。単にエントリアドレスを書き込むだけでなく、より複雑なコードを実行する。特に、前方分岐と後方分岐を行うことが要求される。

難しくて微妙な点を抽象的に話すよりも、私が便利だと思ったワードの例をいくつか挙げてみよう。例によって、あなたは規約を自由に選べますが、基本的な効果は私のやり方と類似になると思います。

IF、ELSE、THENというワードを定義して、次のような条件文の書式を許すようにする。

- ブーリアン値 IF true文 ELSE false文 THEN continue

このワードは、おなじみのALGOLの形式から順列化されていますが、ある種のニモニックな値を持っています。IF、ELSE、THENは命令を生成するワードであるため、このような文は定義にのみ現れることができます。

定義時にIFというワードが実行されます。これは前方ジャンプをコンパイルする。 ここで、議論を横道にそれて、ジャンプを定義しなければなりません。仮想計算機のジャンプ命令は、リテラルと似ています。インラインリテラルは倍長の命令です。前半の疑似エン

トリに対して実行されるコードは、後半の疑似エントリをパラメータとして使用します。ジャンプも同様で、疑似エントリはインラインパラメータを使用して仮想コンピュータの命令カウンタ(IC)を変更します。

このパラメータは、ICに加算する量の正負で、前方ジャンプの場合は正、後方ジャンプの場合は負になります。これは相対的なジャンプアドレスで、この構造は実際のコンピュータで使われているものです。

実際には、条件付きジャンプと無条件ジャンプの2つのジャンプ擬似エントリが必要です。条件付きジャンプはスタックが0でないときだけジャンプし、破壊的な操作(引数は削除される)です。

さて、IFの話に戻ろう。IFは定義時に、条件付きジャンプの疑似エントリをコンパイルし、その後に0を置きます。 そして、この「0」の位置、つまり未知のアドレスをスタックに格納します。スタックは現在使用されていないことを忘れないでください。後で、私たちが定義しているワードによって使われることになりますが、今のところ、私たちは自由にそれを使って、処理を助けることができます。

次にELSEを見てください。定義時には、無条件ジャンプの疑似エントリの後に0をコンパイルしますが、その後、次の利用可能な場所であるDPの現在値をスタック上の場所に格納します。このように、IFで生成される条件付きジャンプの距離を提供する。 実際には、相対アドレスを得るために減算しなければなりませんが、原理は明らかです。そして、そのアドレスの場所をスタックに残します。

最後にTHENを紹介します。これはELSEがぶら下げたままにしていたアドレスを修正する。つまり、DPからスタックを減算し、その結果を間接参照でスタックに格納し、破壊する。このように、IF、ELSE、THENの組み合わせは、スタックを利用して前方ジャンプの仮想計算機命令を構成する。ELSEとTHENは欠落したアドレスを修正する点で同じなので、ELSEはそのまま省略することができる。また、スタックは未実行のジャンプを保存するために使用されるので、IF .... THEN文は入れ子にすることができます。唯一の制約は、すべてのアドレスが決定されること、つまり、すべての場所がスタックから削除されることです。これは、すべてのIFが一致するTHENを持つ場合であり、ELSEは常に任意である。

もちろん、この手法に特別なものはない。すべてのコンパイラは、この方法で前方ジャンプを生成します。やや特殊なのは、これを仮想計算機用の命令のコンパイルに適用することだ。 しかし、これが一番良い方法だと思われる。

関連する構成方法を考えてみよう。ANDの列やORの列で構成される論理式をよく目にする。

このような式の真理値は、式全体が評価される前に決定されることがある。最終的な結果がわかった時点でやめることで、時間を節約することができる。例えば、次のような文を考えてみましょう。

- a b AND c AND IF ... THEN

ここで、a, b, c はブール式であり、この文は ALGOL では次のようになります。

- if a and b and c then ...

もしaが偽であれば、論理和が真になることはありえないので、やめたほうがよいでしょう。もし、この文を次のように書き直すと

- a IF b IF c IF ... THEN THEN THEN

結果は同じになります。a, b, cがすべて真であれば、条件文は実行されます。それ以外は実行されません。各IFは、一致するTHENによって捕捉される前方ジャンプを生成します。IFとTHENを一致させなければならないことに注意してください。実際、これは一種のネストされたIF ... THEN文の一種です。これは非常に効率的な構造です。

- a b OR c OR IF ... THEN

あるいはALGOLでは

- if a or b or c then

もしaが真なら、やめたほうがいい。この文を次のように書き直す。

- a -IF b -IF c IF HERE HERE ... THEN

ここで、以下のように定義すると

- : HERE SWAP THEN ;
- : -IF NOT IF ;

この文は以下のように動作します。a が真なら -IF がジャンプし、b が真なら -IF がジャンプし、c が偽なら IF がジャンプします。最初のHEREはbのジャンプを捕らえ(SWAPはcのアドレスを邪魔にならないようにする)、2番目のHEREはaのジャンプを捕らえ、THENはcのジャンプを捕らえます。こうして、aとbは条件に飛び込み、cはそれを飛び越える。

これは少し不器用な文ですが、これより簡単な解決策は見つかりませんでした。 少しくどいようですが、これ以上シンプルな解決策はありません。 普段から使っていれば、自然と身につくものだと思います。ただ、すべてのIFを一致させることに注意してください。さらに、同じテクニックをより複雑な論理式に適用すると、さらに不器用になります。

### 4.4.4 ループ

定義時に実行されるワードの例について、もう少し続けます。今回は、ループを構成するための後方ジャンプの例です。

BEGINとENDというワードのペアを、次のような文で考えてみましょう。

- BEGIN ... boolean END

BEGINはDPをスタックに格納し、ループの開始の印を付ける。 ENDは、BEGINが残した場所に条件付き後方ジャンプを生成する。つまり、条件付きジャンプの疑似エントリを格納し、スタックから DP+1 を引いて、その相対アドレスを格納する。実行中にブール値が偽であれば、ループ内に留まる。真になると、ループを抜けます。

BEGINとENDは、論理的な条件によって終了するループを提供します。 別のループを定義してみましょう。これは、ループを制御するために、ある範囲のインデックスをカウントするものです。

- a b DO ... CONTINUE

a と b はスタック上の引数を表します。DO は BEGIN と同じように動作します。CONTINUEは、スタック上の上位2ワードが等しいかどうかをテストし、等しくない場合はジャンプする新しい擬似エントリを必要とします。CONTINUEはコンパイル時にこの擬似エントリを格納し、ENDと同様にジャンプ距離を計算します。CONTINUEは、偽ではなく、スタックが等しいかどうかをテストする別の条件付きジャンプを使用しています。これはまた、引数が等しくない限り、非破壊で実行されます。引数が等しくなり、ループが終了すると、引数を削除します。

おそらく、DO ... CONTINUEループの内部で、ループを終了するように引数が変更されているのでしょう。CONTINUEループの内部では、ループを終了するように引数が変更されます。これは、いろいろな方法で行うことができます。 例えば、1 から 10 までのループを実行する場合。

- 10 0 DO 1 + ... CONTINUE

最初の引数は10で、これは停止値である。2番目の引数は0であり、これは直ちにインデックス値である1に増加される。ループ内では、このインデックスが使用可能です。DUP操作はコピーを取得します。インデックス値10でループが実行された後、CONTINUEオペレーションはループを停止し、2つの引数(現在は両方とも10)を削除します。

また、同じループを次のように書くこともできる。

- 11 1 DO ... 1 + CONTINUE

ここでは、インデックスがループの最初ではなく、最後でインクリメントされます。11に到達し、限界の10を超えたら、ループは停止する。

もちろん、ループを逆算することもできますし、その他多くの方法でインデックスを変更することができます。ループは必ず等号で終了する。もちろん、このような柔軟なループ制御は、まったく止まらないという危険性もある。インデックスのインクリメントを間違えると、永久に走り続けることになります。しかし、慎重に使えば、便利なツールである。

DO ... CONTINUEの改良は難しくない。もし引数が等しければ、DOは条件付き前方ジャンプを生成し、CONTINUEはそれを修正することができます。したがって、ループを何度も行うことができる。しかし、このようなループは例外であり、もしループに遭遇したら、それを保護するために必要な条件文が最も厄介であることが分かるでしょう。

### 4.4.5 実装

定義時に実行されるワードの必要性を理解していただけたと思います。また、分岐やループの必要性についてもご理解いただけたと思います。 ラベルについて触れていないことにお気づきでしょうか。私が述べた分岐を生成するワードや、あなたが発明した他のワードは、ラベルのないジャンプを完全に処理することが可能です。HEREの定義で、スタックを操作して、ネストされたジャンプだけでなく、オーバーラップしたジャンプを可能にする方法を見ましたね。しかし、ある意味では、私たちは多くのラベルを持っています。なぜなら、すべての辞書の項目は、実質的には、コードの一部に名前を割り当てているからです。

ここで、私がつまびらかにしたいくつかの問題を考えてみましょう。明らかに、定義中に実行されるこれらのワードを認識できなければなりません。つまり、IF、THEN、BEGIN、ENDなどは、制御ループがそれらをコンパイルする通常のメカニズムを何らかの形でオーバーライドしなければならないのです。実行とコンパイルを区別するスイッチの話をした。同じようなフラグ(1ビット)を各辞書エントリに設け、その値を次のようにしよう。

- 1: 実行

- 0: コンパイル

この値は、スイッチとフラグの両方に適用する。

あるエントリに対して、スイッチとフラグを「or」して、どちらかが1であればそのワードを実行し、そうでなければコンパイルする。

上記のルールは正しく、しかもかなり効率的です。制御ループを効率的にしておきたいということを忘れないでください。そして、実行しなければならないワードがすべてシステム辞書に組み込まれている場合には、このルールは適切です。 残念ながら、上であげた複数の例では適切ではありません。しかし、複雑なことはプログラミングの楽しみの一つです。というわけで、注目してください。私自身もあまりよく理解していない問題を解説してみます。

> 編集部へ：以下、SWAPに関する私の懸念が理解できない。ワード!は耐えられなかった。私が言ったことをすり合わせようとしないでください。できないんです。

私が上にあげたHEREの定義を考えてみてください。

- : HERE SWAP THEN ;

HEREは、これらの命令的ワードの一つです。HEREは、定義時に実行されなければならない。しかし、これは普通の定義として定義されており、コンパイルされることになる。HEREを実行できたとしても、その定義の最初のワードはSWAPである。 次のTHENというワードは問題ない。HEREを実行できれば、THENも実行できるはずだ。しかし、HEREを定義した時点では問題がある。コンパイルするときにTHENを実行しようとするのだ。つまり、命令形のワードをコンパイルしたい場合もあれば、普通のワードをコンパイルしたい場合もある。

では、どうすればいいのか? きっとあなたは、私に解決策があると思っているのでしょう。そのお考えが感動的なところ申し訳ないのですが、私はあまり良いものを持っていません。私の解決策には、「定義の中でリテラルを実行してはいけない」という、小さな、しかし厄介な制約に引っかかってているのです。積極的に言えば、リテラルは定義の中でコンパイルされなければなりません。では、どのように動作するのか見てみましょう。

スイッチSTATEを考えてみましょう。通常は0ですが、":"により1になり、コンパイルを意味します。新しい定義項目":!"を定義してみましょう。":"と全く同じ働きをしますが、例外が2つあります。

- エントリフラグを1にして、命令語であることを示します。
- STATEを2に設定し、すべてのワードを強制的にコンパイルします。制御ループのテストは、STATEとflagが等しい場合に実行されるので、何も実行されません。

";" は変更されず、両方の定義に対してSTATEを0に設定します。 これで、SWAP以外の問題はすべて解決しました。通常コンパイルされるはずのワードをどのように実行するか?

新しいエントリ "!" を定義します。これは最後にコンパイルされたエントリを実行させ、コンパイルから外す。これで、HEREの定義を次のように書き直すことができる。

- :! HERE SWAP ! THEN ;

と書けば、うまくいくでしょう。ルールを見直してみる

- すべてのワードが正常に実行されます。
- 定義では、命令形と判定されたワードのみが実行されます。
- どのワードも「！」をつけることで命令形にすることができます。
- 定義に「:」を使う代わりに「:！」を使うことで、定義を命令型にすることができる。

ここで、先ほどの制約を確認しておきましょう。なぜなら、リテラルはダブルレングス命令であり、「！」コードはそれを知る由もないからです。まあ、最後にコンパイルした命令の長さを示すフィールドを設定すればいいのですが、それほど大きな問題ではありません。それに、この場合、連続した!sは機能しません。

# 4.5 コードエントリ

定義と、仮想計算機のための命令をどのように実質的にコンパイルするかについて説明しました。では、実際のコンピュータのためにコードをコンパイルするのはどうでしょうか。もちろんできます。しかし、おそらくやろうとは思わないでしょう。

基本原則が邪魔をするのです。プログラムにコード項目を追加すると、非常に大きなパワーと柔軟性が加わります。コンピュータができること、コンピュータが持っている命令、コンピュータのハードウェアを使ったトリックなど、何でも指先一つでできるようになるのです。これはとても素晴らしいことなのですが、このようなパワーが必要になることはほとんどない。 また、その代償は大きい。有用なコンパイラを提供するためには、多くのエントリ(例えば10項目)が必要であり、さらにすべての命令のニーモニックが必要です。さらに、コードをコンパイルする問題に特化したアプリケーション言語を設計しなければならない。

そのような努力の可能性や価値を低く評価するつもりはありませんが、あなたは最初に何らかの言語でプログラムを書きました。もし、追加のコードが必要なら、プログラムを再コンパイルして必要なものを追加する方がずっと簡単です。オーダーメイドのコードを必要とするアプリケーションがある場合、あるいは異なるユーザに異なるコードを提供することによって利益を得ることができる場合、あるいは異なる時間に異なるコードを提供することによって利益を得ることができる場合にのみ、基本原則を満たすことができるのです。

一方、コード入力から始めると、これまで述べてきた算術演算子、名詞入力、定義など、他のすべての入力を構築することができるようになります。第9章では、コードエントリを本当に重要な役割で使用し、他の方法よりもはるかに効率的で強力なプログラムを実現する方法を紹介します。しかし、それを除けば、コードエントリがあまり重要にはならないと考えています。

では、どのようにしてコードを生成するのでしょうか。まず、コードエントリを定義する定義項目が必要です。コードエントリの特徴は、そのパラメータフィールドに格納されたコードを実行することです。したがって、定義エントリ(CODEとします)からENTRYに渡されるアドレスは、最初の命令を置く場所でなければなりません。これは、エントリ自体にスペースがあるため、DPではなく、DPに定数を加えたものになります。

次に、DPに数値を格納するためのエントリが必要です。このようなルーチンを何度か使い、変数や定義を構築してきましたが、そのためのエントリはありませんでした。出力項目と矛盾するかもしれませんが、","というワードを提案します。スタックからパラメータフィールドに数字を移動させるだけです。命令はもちろん数字です。スタック上に構築して、それを格納するのです。 ちなみに、コードのコンパイル以外にも、これは便利なエントリです。あらゆる種類のデータ配列を初期化するのに役立つことがわかるだろう。

さて、先ほどの注意の意味がお分かりいただけたでしょうか。今まで直接参照する必要のなかった、プログラムにコンパイルされたコードにアクセスするエントリを用意しなければならないのです。例えば、RETURN です。ルーチンが終了したら、組み込みエントリと同じように制御ループにジャンプしなければなりません。しかし、コア内の制御ループの位置はわかりませんし、プログラムの変更に伴って制御ループも移動します。そのため、RETURN命令を生成するためのエントリが必要です。

同様に、定義エントリをコンパイルする場合は、ENTRYへのサブルーチン呼び出しを生成するエントリを用意しなければなりません。他のコードは WORD や NUMBER、あるいはプログラム内で既に利用可能な機能にアクセスしたいかもしれません。さらに、使用するフィールドのための変数エントリを定義する必要があります。出力におけるDとFも、お

そらく、STATEとBASEも用意する必要があります。基本的に、問題は、プログラムの内部ですでに利用可能なすべてのラベルを、プログラムの外部で利用可能にしなければならないことです。その努力を正当化するために、十分に使用しなければなりません。

よし、そこまではできた。さて、次は命令をどのように構成するかを決めなければなりません。命令、インデックス、アドレスなど、いくつかのフィールドがありますが、これらを別々にスタックに載せて、どうにかして組み合わせたいところでしょう。これは簡単なことですが、設計が難しいのです。おそらく、アセンブラをコピーしたくはないでしょうし、どうせそのフォーマットを都合よく追いかけることはできないでしょう。実際、読みやすいコンパイラ言語を設計することは可能ですが、それなりの努力が必要です。 定義は必要な道具をすべて提供してくれます。

例えば、ある命令とアドレスを "or "で結び、それを格納するような定義を書くことができます。また、ハードウェアが不格好な場合、絶対アドレスを相対アドレスに変換する定義や、適切なページング制御を提供する定義も可能です。必要なもの、欲しいものは何でも簡単に定義することができる。このようなコンパイラは、適切に作られたものであれば、それ自体が重要なアプリケーションとなりますし、もしそれを行うのであれば、必要な時間と労力を費やす覚悟をしてください。

仮想計算機の条件文とループについて説明しました。ハードウェアのばらつきを考慮した上でも、全く同じ手法が適用できます。実際、私はもともと実機のコードにスタック指向の分岐生成機能を適用していました。このような記述は、まさにアセンブラとコンパイラの違いである。基本原則を忘れるな。

コンパイラの有用な使い方として、新しい種類の名詞の定義を可能にすることがある。つまり、新しい定義項目を作ることである。例えば、プリミティブコンパイラを使って、先に述べたような命令項目を定義することを考えてみてください。あるいは、スタックの先頭を定数倍するエントリを定義したいかもしれません。

能力を追加する場合、通常、いくつかの異なるエントリが協力して提供される必要があります。この場合、 `ENTER` と `;CODE` です。説明しましょう。

- : UNIT ENTER , ;CODE 1 V LDA , SP MPY , SP STA , NEXT ,
- 2.54 UNIT IN
- 4. IN

最初の行では、UNITというワードを定義しています。次の行は、この定義項目を使用してIN (インチ) というワードを定義しています。最後の行は、IN を使って 4 インチをセンチメートルとしてスタックに載せています。この3行は次のコード

- : IN 2.54 * ;

と等価であり、確かにこちらの方がシンプルです。しかし、多くのUNITを定義したい場合は、特別な定義項目を用意した方がはるかに便利で効率的です。

最初の特別なワードは ENTER です。これはすべての定義エントリで使用される ENTRY サブルーチンを呼び出しますが、実行されるコードの場所として 0 アドレスを渡します。UNIT の定義を見てください。ENTER というワードは命令形です。前半は擬似エントリ、後半は0定数という倍長の疑似命令を生成します。実行時に、疑似エントリはENTRY を呼び出して新しい辞書エントリを構築し、実行されるコードのアドレスとして次の定数を渡します。;CODE というワードは、";"と CODE を組み合わせたものである。これは、UNIT の定義を終了させ、DP を ENTER によって確立されたアドレスフィールドに格納する。ENTERは、それを使用する定義の最初のワードであるように制限されているので、CODE はDP を格納する場所を知っています。

ENTERの位置に関する制限は重要ではなく、他の場所と同様に最初のワードであってもよい。UNIT の場合、定数を格納するための","だけが必要であった。他の名詞は、パラメータ・フィールドを確立するために、もっと複雑な処理が必要かもしれません。

CODE に続くコードの例を挙げました。命令のニーモニックとアドレスが「,」で寄託されているのがわかりますね。このコンパイラ言語については、あなたのコンピュータには関係ないので、説明するつもりはない。

もう一つの提案は、役に立つかもしれない。新しい種類の定数、すなわち命令を定義することができます。命令が実行されると、スタック上のアドレスが期待され、そのパラメータ・フィールドから定数を取り出し、命令の構築を完成させてそれを格納します。 おそらく多くの命令があり、多くの数を使用することでしょう。そうすれば、多くの格納エントリを節約することができます。

申し訳ありませんが、例を挙げるのは無理があると思います。もし、私がすでに述べたことから、独自のコードエントリを構築する方法がわからないのであれば、それは忘れてください。このアプリケーションは非常にマシンに依存しており、当然そうなります。同じコードを複数のコンピュータに適用しようとしないでください。(通常の)定義がすでにそれを行っています。コードの目的は、あなたの特定のコンピュータの特性を利用することです。

# 5. メモリを持つプログラム

あなたは、必要な量の入力を提供しなくても成長するプログラムの価値を認めるかもしれません。当然ながら、100の辞書項目があっても、その一つ一つを入力しなければならないのでは、ほとんど意味がない。エントリを保存する場所が必要であることは明らかであり、かつ、その場所はディスク(またはドラム、または他のランダムな二次記憶)であることも明らかである。

しかし、どのように項目を保存するかは明らかではない。修正できない状態でディスクに保存しないことは第二の原則であるべきですが、このルールは普遍的に無視されています。辞書の項目を単純にコピーすることは、もう1つの基本原則である「コアアドレスをディスクに保存しない」に違反する。すべてのコードアドレスを追いかけなければ、プログラムを修正することはできないのだ。

幸いなことに、解決策がある。辞書エントリを構成するテキストをディスクに保存するのである。入力ルーチンをメッセージ・バッファの読み込みからディスクの読み込みに変更するのは簡単なことである。この章では、その方法を紹介する。

## 5.1 ディスクの構成

ディスクを組織化する方法はただ一つである。コアが多数のワードに分割されるのと同じように、ディスクも多数のブロックに 分割されなければならない。ワードがコアから取り出せる最小のフィールドであるのと同じように、ブロックはディスクから取り出せる最小のフィールドである。1ブロックには256個のワードが含まれる。

1ブロックが256ワードであるのは、1バイトのアドレスの大きさであり、4バイトのワード256個で1024バイト(通常の表示画面(scope)で表示可能なテキスト量)を保持できるからである。

しかし、ここでもアプリケーションとハードウェアが主役になる必要があります。ディスクは通常、ハードウェアが有利なブロックサイズを持っています。その倍数を選択する必要があります。 アプリケーションでは、データをディスクに保存することもあるでしょうから、データだけでなくテキストにも有効なブロックサイズを選択しなければなりません。512文字以下、1024文字以上とします。最近、128ワードブロックが話題になりましたが、6バイトや3バイト(文字)のワードなら問題ありません。

### 5.1.1 ブロックを得る

ランダムファイルの構成を考える上で、ある種の原則は明白である。ブロック間の相互参照は、おそらく必要でしょう。このような参照は、絶対ブロックアドレスを使用すれば簡単ですが、そうでない場合は非常に不格好です。ブロックを動かさないと約束するならば、絶対アドレスを使ってもよい。つまり、ディスクのパッキングは絶対にできないということだ。どうせディスクのパッキングは望まないので、快く受け入れた。

つまり、ブロックの中のデータが使えなくなると、ブロックサイズの穴が空くことになる。この穴を何とかして再利用しなければならない。つまり、ディスクをブロックサイズに分割して割り当て、再割り当てしなければならないのです。

ブロックのアドレスも含めて、すべてのアドレスは0から始まります(そうしないと、永遠に1を足したり引いたりすることになります)。しかし、ブロック0は何にでも使えるわけではありません。アドレスの間違いのほとんどは、ブロック0に関係しています。ブロック0をときどき見てみると、驚くような発見があります。ブロック1は、実行ごとに覚えておく必要があるものを保存するのに便利な場所であることがわかるでしょう。例えば、再利用可能な最初のブロックのアドレス(なければ0を格納)、最後に使用されたブロックのアドレス(初期値は1)などです。

保存するために、ディスクを(別のディスクやテープに)コピーしたくなるだろう。 コピーするのは使用したブロックの数だけでよく、通常ディスク容量の半分以下である、さもなければ容量がかなり心配になる。 もしブロック1を破壊してしまったら(破壊してしまう)、バックアップからディスク全体を再ロードしなければならない。決してブロック1だけを復旧させようとしないでください、恐ろしく混乱してしまいます。

このディスクにオブジェクトプログラムを入れることもできます。いいでしょう! 何ブロックも使うことはないでしょう。初期ロード(ブートストラップ)を行うためにブロック0から開始する必要があるかもしれません。でも、ブロック0を破壊してしまうので、バックアップからプログラム(だけ)を再ロードできるようにしましょう。空き領域情報のあるブロック(ここではブロック1と呼ぶ)を破壊した場合のみ、データ(全データ)を再ロードする必要があります。多くのブロックを破壊しない限り。最小限の努力ではなく、最小限の混乱の道を選びましょう。ディスクの再読み込みはあなたを混乱させ、あなたは何を変更したかを忘れ、それを発見するのに何日もかかるでしょう。それよりも、何時間もかけてテキストを再入力し、データを再入力する方がずっといい。

そのため、ブロックが必要なときは、ブロック1を読むワード(GET)をタイプし、再利用のためにブロックをスタックに上げ、そのブロックを読み、その最初のワードの内容をブロック1に置き、ブロック1を書き直します。もちろん、最初のワードには次の再利用可能な

ブロックのアドレスが含まれています。再利用可能なブロックがなかった場合(最初はそうでした)、GETは最後に使用したブロックをインクリメントし、スタックに載せてブロック1を書き直します。その後、GETは新しいブロックをゼロクリアし、再書き込みします。

いくつかのコメントがあります。GETはその結果をスタックに置くこと、つまり論理的な場所に置き、そこでさらに利用できるようにすることに注意してください。ブロックは、使用するディスクを拡大するよりも、再利用されることに注意してください。これは、(ディスクの)アームの動きの問題を除けば、理にかなっている。アームの動きは忘れてください。うまくやってゆくしかないのです。これは結局のところ、ランダムメモリなのです。ブロックを0にクリアするのを怠ってはいけません。

### 5.1.2 ブロックの解放

ブロックを解放するには、ブロックをスタックに載せて RELEASE と言ってください。これはブロック 1 を読み、再利用のために次のブロックを取り出し、スタックをそこに置き、ブロック 1 を書き込みます。その後、リリースされたブロックを読み、古い次のブロックを最初のワードに配置します。もちろん、私たちが行っているのは、GET が使用する利用可能なブロックの連鎖を構築することだけです。あなたが解放したブロックは、他のブロックとリンクしている可能性があります。それらもすべて解放しなければなりません。便利な方法は、最初のワードをリンクフィールドとして使うことです。そうすれば、利用可能なブロックチェーンは、他のブロックチェーンと同じになります。チェーンを連結するには、最初のブロックをブロック1に置き、チェーンを最後のブロック(リンクの0)まで下げて、古い次のブロックをそのリンクに置きます。

利用可能なブロックの数を管理しようとは思わないでください。そのような面倒なことをする価値はありません。どうしても知りたい場合は、使用可能なチェーンの長さを数えるとよいでしょう。

ブロックの種類が十分ある場合は、ブロックを識別するコードを最初のワード(または2番目)に保存しておくと便利です。そうすれば、ある種のブロックはすべて調べることができます。利用可能なブロックはコード0であるべきです。

ブロックの数は、おそらくディスクによって制限されているが、ブロックのアドレスを格納するフィールドによって制限される場合もある。注意! 最初の制限は、読み出しサブルーチンを修正して、いくつかのディスクの中から1つを選択することで回避できます。2番目の制限は、すべてのブロックアドレスを再フォーマットする必要があります(ディスク上の相互参照、覚えておいてください)。

### 5.1.3 ディスクの読み書き

ディスクの読み方はご存じだと思います。しかし、少しでも困難が生じるようなブロックサイズ、例えばトラックをまたぎブロック半分の長さは選ばないようにしましょう。GETルーチンをチェックすると、コアに同時に2ブロック必要であることがわかります。これは合理的な最小値で、あるブロックから別のブロックへ物を移動させるのが簡単になります。しかし、コアには多くの余剰があり、特にアクセス時間が顕著な場合は、ディスクのバッファリングに使用した方がよいでしょう。

どのブロックがコアにあるのかを指定するテーブルが必要でしょう。

しかし、ブロックを変更したときに(直ちに)それを書き込むべきではありません。むしろバッファテーブルで'書き込むべき'マークを付けます。そのバッファを再利用するときは、古いブロックを先に書き込みます。原理は、一度変更したブロックは再び変更する可能性が高いということです。書き込みを最小限に抑えれば、多くのディスクアクセスを節約することができます。もちろん、トレードオフがあります。プログラムがクラッシュした場合、コア内のブロックが更新されているがディスクに反映されていない可能性があります。プログラムを再スタートして、コアバッファを維持することができるはずです。

もちろん、複数のコアバッファは、割り当ての問題を意味します。単純なラウンドロビンは、他の方式と同様に効果的です。

データを順次読み込む場合、連続したブロックを同時に読み込むことで、多くのアクセスを節約することができます。しかし、順次読み出しの中にランダムな読み出しが混在する可能性がある。効果的な解決策は、シーケンシャル領域の最後のブロックとブロック数を読み取りサブルーチンのどこかに保存することです。ブロックがコアでなく、シーケンシャル領域内であれば、連続したバッファがある限り、連続したブロックを読むことができます。これ以上、つまり利用可能なバッファを増やそうとしないでください。正味のところ、干渉の制約下で、シーケンシャルブロックに対してできる限りのことをすることになります。

ディスクの読み書きに多くの労力を費やすことは避けられないでしょう。しかし、基本原則を忘れないでください。

## 5.2 ディスク上のテキスト

ディスクには何百ブロックもの大量のテキストが保存されますが、これはおそらくディスクのごく一部でしょう。残りの部分は、おそらくアプリケーションのデータです。

テキストを含むブロック(プログラムで読み込んで実行するテキストという意味)は、1つの長い文字列を含んでいます。最初のワードが制御情報を含んでいる場合、それは2番目のワードから始まり、特定のワードが終わりを示すまで続きます(おそらく;S)。この終了ワードが重要なのは、入力ルーチンにブロックの終了をテストさせるのは不都合だからです。このワードを省略してはいけないことはすぐにわかるでしょう。

テキストを含むブロックには特別な名前をつけるべきです。なぜなら、会話の中で頻繁に使うからです。私はこのようなブロックを、テキストが一枚の紙を埋めることから「シート」、テキストがスコープのスクリーンを埋めることから「スクリーン」と呼んでいます。READ というワードを定義し、入力アドレスと、次にスキャンする文字のブロック位置と文字位置がリターンスタックに保存され、入力ポインタがスタック上のブロックと最初の文字位置にリセットされるようにしてください。元の入力ポインタに戻すには ;S というワードを定義します。非常に簡単に、ブロック 123 を読み込むことができます。

- 123 READ

しかし...常に「しかし」がありますね。スキャンする前に現在のブロックを読むようにワードルーチンを変更しなければなりません。これは高価ですが不可欠です(もちろん、ブロックがコアにある場合は実際の読み取りは行われません)。これは、あるスクリーンが他のスクリーンの読み取りを指示した場合に特に発生する可能性があります(そのようになります)。この問題に対する他の解決策は満足のいくものではありませんでしたので、コードを飲み込んでください - それは大きなものである必要はありません。

ディスク上のテキストでは、「入力」が少量であるという本来の性質が損なわれていることに気づくでしょう。多くのワードを読み、多くの辞書検索をすることになる。しかし、マイクロ秒のコンピュータでは、それに気づくことはないでしょう。

## 5.2.1 テキスト編集

修正できないものはディスクに入れないこと! そして、そもそもどうやってテキストをディスクに入れるかについては、まだ議論していません。カードから読み込まないでください。カードから読み込むと、カード読取に労力を割くことになりますし、どうせカードに穴を開けなければならないのですから。タイプしてください。ブロック(SCREEN)に格納されたテキストを編集するために必要な定義は簡単です。

引用符で囲まれた文字列を扱えるようにすることである(4.1)。それを踏まえて、テキスト編集画面を表示することにする。これは、定義の価値を示す単純な例である。これは私が示す自明ではない例の最初のものである。もう、ちゃんと見てみようという気になるはずだ。

当然ながら、あなたはこれらの定義を2回タイプする必要があります。一度は辞書に登録し、もう一度はそれを使って画面に表示させる(ブートストラップ)。実際には何度も入力することになるでしょうが、2回が最低ラインです。

あるプログラムで使用したEDIT画面の注釈付きコピーを示します。この画面では、意味がよくわからないシステムエントリが使われています。これらは、アプリケーションの他の側面から借用したものです。

- 0 C1 42 # :R RECORD

ここでは、フィールドの説明を構築している。RECORDは、現在のブロックのワード0の文字1から始まる42文字のフィールドである(理解できる)。42文字の行を15個保持できるブロックを使っています。1ワードは6文字なので、7ワードの行を15個保持できることになります。

- : LINE 1 - 7 * RECORD + ;

ここでは、行番号(1-15)をフィールドアドレスに変換する動詞を定義しています。これは、ワード指定(下位ビット)を変更することによってRECORD記述子を修正する。したがって、行1はワード0から始まり、行2はワード7から始まる...といった具合です。

- : T CR LINE ,C ;

`3 T` とタイプしたら - 3行目をタイプして欲しい。Tはキャリッジリターン(CR)を行い、LINEを実行してフィールドアドレスを計算し、その(文字)フィールドをメッセージバッファ(,C)にコピーします。

- : R LINE =C ;

もし私が `" NEW TEXT" 6 R -` とタイプしたら、6行目を引用符で囲まれたテキストに置き換えたいのです。先頭の引用符は、文字列記述子をスタックに置く。R は次に LINE を実行し、その後に =C を実行して引用符付きの文字列をフィールドに格納します。ブロックは変更されたので、自動的に書き込まれる。

- : LIST 15 0 DO 1 +
- CR DUP LINE ,C DUP ,I CONTINUE ;

LISTはブロック全体をリストアップします：1行42文字の15行とその後ろに行番号が続きます。スタックを1〜15まで変化させながら、DO-CONTINUEループを設定します。ループの各回で、CRを実行し、スタックをコピーしてLINEを実行し、フィールドをタイプし(,C)、再びスタックをコピーして整数としてタイプし(,I)ます。

- : I 1 + DUP 15 DO 1 -
- DUP LINE DUP 7 + =C CONTINUE R ;

もし私が `" NEW TEXT" 6 I` とタイプしたら、テキストを6行目の後に挿入したいのです。"I"は、まず7行目から14行目までを1つ下にずらし(15行目を失う)、次に7行目を置き換える必要があります。行番号に1を加え、14から始まる逆方向のDO CONTINUEループを設定し、2つのフィールド記述子 LINE と LINE+7 を構築し、それらをシフトします (,C)。ループが終了したら、Rを実行する。

- : D 15 SWAP DO 1 +
- DUP LINE DUP 7 - =C CONTINUE " " 15 R ;

12 D - と入力すると、12行目を削除したいのです。Dは13-15行目を1つ上に移動させ、15行目を消去しなければならない: スタック+1から15までDO-CONTINUEループを設定する。各反復は、フィールド LINE と LINE-7 を構築し、それらをシフトします (=C)。そして、15行目を空白で置き換えます。

これだけです。10行のコードでテキストエディターを定義することができます。これは最も効率的とは言えませんが、十分に速く、多くの点を説明しています。扱うテキストの量が少なければ、賢さは必要ない。

これだけです。10行のコードでテキストエディタを定義することができます。最も効率的とは言えないが、十分に速く、多くの点を説明している。少量のテキストを扱う場合、賢いやり方である必要はなく、機械に仕事をさせればいいのです。LINEという動詞は非常に便利なものです。このような便利な動詞は必ず経験的に発見されるものです。動詞 ,C と =C はこのメソッドの中心である。ちなみに、これらは64文字未満のフィールドに対してのみ機能する。ある定義が別の定義を参照したがることに注目（RはIとDが使用、LINEはすべての定義が使用）。IとDが似ているようで異なることに注目してください。また、いくつかの動詞を使うことで、多くの情報管理が不要になり、細かいことではなく、問題に集中できることに注目してください。

# 6. 出力を持つプログラム(Programs with output)

ここまでで、プログラムの心臓部が制御ループであることはお分かりいただけたかと思います。制御ループは、動作だけでなく、プログラムの哲学や構成も制御します。ワードを読み、辞書で見つけてコードを実行し、失敗したら2進数に変換してスタックに置き、失敗したらエラーメッセージをタイプします。

これまでのところ、私はこのエラーメッセージを無視してきました。それは、それが重要でないとか、実装するのがつまらないからではなく、出力という難しい課題の一部だからです。論理的には、出力についてここまで引っ張るべきではなかったのです。制御言語であっても出力は必要だからです。しかし、このプログラムではいつものように、出力は、今まで説明したばかりの他の機能に関わっていました。これから説明する出力機能のうち、あなたのアプリケーションが必要とする機能を実装するのは、あなたの責任に委ねます。

ほとんどのコンパイラ、ひいてはほとんどのプログラマは、出力を入力の逆とみなしています。例えば、FORTRAN は入力と同じ FORMAT ステートメントを出力に使用し、2 つのプロセスが非常に似ていることを示唆しています。しかし、そうだろうか?

入力を構成する：ワードを選び、それらを組み合わせてかなり複雑なフレーズを作る。プログラムはこの入力を解読し、その意味を抽出するのに多大な労力を費やす。応答に対しては、そのような入念な手続きは一切行いません。出力のほとんどが「OK」というワードで構成されているのがわかるだろう。あなたはコンピュータに話しかけているのですが、コンピュータはあなたにほとんど話しかけることはなく、せいぜい唸っているに過ぎないのです。

私は、この2つのプロセスには共通点がなく、あなたが入力を準備するのと同じようにコンピュータが出力を準備するわけではないと主張します。第8章では、あなたのプログラムが複雑な出力メッセージを構成する方法について説明します。このようなテクニックは双方向の対話を提供するかもしれませんが、入力を解釈することとの類似性はさらに低くなります。

# 6.1 出力ルーチン

出力サブルーチンは3つ必要ですが、2つでもよいかもしれません。1つは、指定された位置から何文字かをタイプするもの(TYPEN)。1つは、スペースに遭遇するまで文字を(そのスペースも含め)出力するもの(TYPEB)。この最後のものは、辞書の形式に依存し、エントリのワードを入力するために使用されるからである。 もちろん、これらは入力に使用する取り出しと格納のサブルーチンを使用する必要があります。

エラーメッセージの組み立てを例にして説明しましょう。あなたは今、入力メッセージを入力し、キャリッジは最後の文字として配置されています。まず、スペース1文字が欲しい。次にTYPEBを使って、現在のワードを印字します。これでエラーが発生し、どこでエラーが発生したかが分かります。 バッファなしのデバイスではこれは必要ありません。その後、もう一度TYPEBを使って、エラーを説明するワードを印字してください。長いエラーメッセージは避けましょう - タイプされている間、待つのはあなたです。あなたは多くのエラーを検出することができますので、それらを生成するルーチンを考案する価値があります。

エラーを発見したら、もちろんそれまでやっていたことをやめる。訂正してまた始められるように待機しているのに、続けようとしても無駄です。しかし、どうせリセットしなければならないようなものは、リセットしておくと便利です。特に、スタックを空にしておくことです。パラメータスタックがあれば、エラーの場所を特定できるかもしれないので、これは時に不幸なことです。しかし、通常は最も便利な方法です。 辞書のリセットは、何をリセットしたいのかよく分からないので、試さないでください。

# 6.2 応答(Acknowledgement)

第3章では、メッセージを送受信するためにサブルーチンを書かなければならないと述べました。ここで、これらのサブルーチンをどのように使うべきかを正確に説明する必要があります。

入力と出力が同じメッセージバッファを共有していることを思い出してください。このことは、今ではトラブルの原因になっています。しかし、これにより7章の強力なメッセージルーチンをかなり単純化することができます。バランス的には単一のメッセージバッファが最適のようです。

まず、メッセージを送信するサブルーチンをSENDと呼ぶことにします。これは1行を送信し、行にキャリッジリターンを追加し、その他必要な制御文字と必要に応じて文字を変換する必要があります。 メッセージを受信するルーチンはQUERYです。これはルーチン

であり、サブルーチンではありません。QUERY は SEND を呼び出してメッセージを送信し、入力メッセージを待って処理します。制御文字を除去し、必要に応じて文字を変換します。入力ポインタ IP を初期化し、NEXTW にジャンプします。プログラムは、SEND を使って好きなところに出力を送れることに注意。しかし、入力は出力と同時に QUERY でしか受け取れない。出力を挟まずに連続したメッセージを受信する機能はありません。これはまさにあなたが必要とする動作であり、実際にメッセージ I/O のコーディングを簡素化するものです。

では、QUERY の使い方を説明しましょう。各入力メッセージは、空白で囲まれた非印刷文字である End-of-message word で終了します。このワードは、OK というワードをタイプして QUERY にジャンプする辞書エントリを持っている。このように、プログラムは各入力メッセージを解釈した後、「OK, message received and understood」という短い了承メッセージを印字し、次の入力を待ちます。

入力メッセージが出力を生成した場合、それ自体を破棄することに注意してください。 つまり、入力がすでにあるにもかかわらず、出力はメッセージバッファに置かれます。したがって、出力を生成するワードは、メッセージの最後のワードであるべきで、それ以降のワードは見られないからです。特に、メッセージの終わりのワードは見られず、返信の OK も打てません。これは、あなたが望んでいることです。OK は、他の出力の代わりにタイプされるだけです。

OK は入力メッセージと同じ行に表示され、最後のワードとは少なくとも 1 つのスペースで区切られていなければなりません。QUERY は、ほとんどのタイムシェアリングシステムがそうなのですが、キャリッジリターンでメッセージの受信を確認すべきではありません。解釈完了時の OK が唯一の承認となります。OK を同じ行に置くと、出力と入力を区別しやすくなり、会話も圧縮されます。これは、限られた大きさの表示器においては特に価値があります。ユーザは出力を受け取るまで入力をタイプしてはならない。この規則を強制することが重要なのは、マルチユーザ・プログラムだけです。これについては第 7 章を参照してください。

メッセージバッファに入力があるかどうかを判断するために、EMPTY というフィールドを設けます。QUERY は empty を false に設定し、各出力生成エントリはそれを true に設定する必要があります。実際に出力生成動詞は互いに共通点が多く、それぞれ以下のようなルーチンにジャンプするはずです。

- スタックを削除する。各出力動詞は引数を持たなければならない。最後の引数はこの時点でドロップすることができ、スタックポインタはその下限に対してチェックされます。

- EMPTY を真に設定する。
- NEXT が NEXTW を含み、SCREEN が 0 であれば、QUERY にジャンプする。 この状況では、メッセージバッファにそれ以上の入力はありません。
- NEXT にジャンプします。

入力が定義や画面から来る場合は、メッセージバッファとの間に矛盾が生じないことに注意してください。メッセージバッファから入力が読み込まれている場合のみ、問題が発生します。

しかし、入力のソースが変更される箇所が 2 箇所あります。これは、「 ; 」と「 ;S 」のコードにあります。もし「 ; 」が NEXTW を NEXT に戻すなら、入力が可能であることを保証しなければならない。同様に、「 ;S 」が SCREEN を 0 に戻す場合、EMPTY が真であれば QUERY にジャンプしなければならない(NEXT は NEXTW であることが保証されている)。

必要なロジックは図 6.2 にまとめたとおりで、メッセージバッファの二重化の代償と言えます。最後の複雑な問題は、EMPTY に関するものです。真であれば、入力が破壊されたことを示しますが、出力が現在メッセージバッファにあることを示すわけではありません。出力はメッセージバッファに置かれ、すでに送信されている可能性があります。メッセージバッファが空の場合は、QUERY に移る前に OK と入力してください。

# 6.3 文字列

何事も簡単にはいかないものですが、この機能は私の宿敵です。 おそらく、この機能の価値を測る尺度は、その実装の難しさにあるのでしょう。 文字列というのは厄介な存在です。ほとんどの場合、それを置く場所がないからです。数値リテラルは最も自然な形でスタックに格納されます。文字列はスタックに収まらないし、そもそも文字列をどうしたいのかがわからない。

私の解決策はこうです。文字列を見たら、そのままにしておくのです。 最初の文字列のアドレスと、文字列の文字数を示す記述子をスタックに置く。文字列を読み飛ばす。つまり、入力ポインタをその端まで進める。もちろん、この順序で実行することはできません。なぜなら、スキップすることによってのみ、文字数を知ることができるからです。

文字列はどのように見えるのでしょうか。あらゆる方法の中から、ひとつだけ完全に自然な方法を選びましょう。

- "ABCDEF ... XYZ"

文字列は引用符で囲まれ、引用符以外のあらゆる文字、特に空白を含むことができます。

すぐにトラブルになりますね。文字列はどのように見分けるのでしょうか? もちろん、先頭の引用符で。しかし、その引用符を認識するようにワードのサブルーチンを改造しているだろうか? そうすると、リーディングクオートを他の目的に使うことができなくなる可能性がある。それよりも、引用符はそれ自体がワードであり、他の辞書項目と同様に扱われ、再定義が可能であることが望ましいのです。しかし、ワードは空白で終わるので、引用を例外とすることにはまだ抵抗があります。そこで、文字列を入力してみよう。

- " ABCDEF ... XYZ"

余分なスペースは煩わしいものですが、第8章では、今挙げたような異議を唱えることなく、それを排除する方法をお伝えします。つまり、文字列はクォートスペースで始まり、クォートで終わります。

文字列はそのままにしておき、単にそれがどこにあるかを記憶しておくだけであることを忘れないでください。私たちは入力バッファの中の文字列について話しているのですから(今のところ)、出力や追加入力で文字列を破壊する前に使った方がいいのです。いつ破壊されるかはいろいろなことに左右されるので、すぐに使うのが一番のルールです。

文字列で何ができるのか? 私は2つの使い方しか見つけられませんでした。それらは非常に似ていますが、似ていることを利用するのが実装のもどかしさの一つです。文字列を印字するか、文字フィールドに移動させるかです。

文字列を印字するのは簡単です。TYPENサブルーチン用のパラメータを設定するために、スタック上の記述子を使用するエントリを定義します。

文字列を移動させるのは難しいですが、それでも簡単です。スタック上に 2 つの記述子があります: 上はフィールド記述子、下は文字列記述子です。入力と出力のポインタを設定し、2つのフィールドサイズのうち小さい方の長さの文字を移動させます。移動先のフィールドの残りをスペースで埋めます。このとき、移動できる文字数以上の文字を移動してはいけないことに注意してください。もちろん、文字列ディスクリプタが正しいサイズであることはほとんどありません。 文字列の切り捨てはエラー条件ではありません!

以上のことができれば、ある文字フィールド1つを別の文字フィールドに移動させることもできます。つまり、文字列とフィールドの記述子に互換性を持たせれば-これが面白さを増します。フィールドを文字列に移動させないようにしたいかもしれませんが、そんなことはどうでもいいでしょう。

問題は、上記のすべての要求を調和させることです。最適なコードを作ることではなく、サイズ、スピード、制約、正しい動作など、少しでも許容できるコードを作ることです。

フィールド記述子の話にスライドしてしまいました。文字フィールドを印字したいと思うかもしれませんが、当然ながら文字列記述子と同じコードが動くはずです。

## 6.4 フィールドエントリ

これまで、さまざまな種類の数字が必要なこと、そのために必要な項目が異なることをお話ししてきました。しかし、これらの項目はすべて計算を扱ったものです。もうひとつのエントリは、より洗練された出力のために有用です。最も一般的な用途は、データレコードのフィールドを定義することなので、私はこれをフィールドエントリと呼んでいます。

変数に関連する記述子に加えて、フィールドエントリには出力形式を指定する追加のパラメータが必要です。出力するフィールドの幅を一度に指定でき、すべてのレポートで自動的にそれを使用できるのは非常に便利である。また、フィールドの名前を参照できるのも便利です。もちろん、フィールド名は辞書エントリに含まれています。

ですから、フィールドエントリはそれ自身のアドレス、つまり辞書エントリをスタックに置くというのが便利な慣例になっています。変数エントリでは、変数のアドレスがスタックに格納されることを思い出してください。エントリの名前が知りたければ、このアドレスはそれがどこにあるかを教えてくれる。書式が必要な場合は、このアドレス(定数でオフセットされている)が、書式がどこにあるのかを教えてくれます。そして、フィールドのアドレスが欲しければ、それも手に入れることができます。これは、変数に対して自動的に実行される処理です。

これらの様々な機能を実現するためには、様々な項目が必要です。定義しておくとよいでしょう。

- ,NM - フィールド名をタイプアウトします。
- F - フィールド幅を抽出します。
- @F - フィールドアドレスの取得

@F を @ と互換性のあるものにすることができるかもしれません。あるいは、@をフィールドエントリに対して自動的に正しく動作するようにする。変数のアドレスとフィールドエントリのアドレスを区別したい場合があります。これは、数字の種類を区別するのと同じ理由で、同じ操作(この場合、おそらく@と=)がすべてに対して機能するようにするためでしょう。

基本原則を適用しましょう。

# 7. Programs that share

当たり前ではないのですが、これまで述べてきたように構成されたプログラムは、複数のユーザを同時に扱うのに理想的に適しています。対話型処理の基本的な問題はすべて、一人のユーザとの対話によって解決されている。すべてのデータはユーザの辞書に格納されているか、格納することができるような構成になっています。ユーザを区別するためには、単にプログラムが適切な辞書を認識する必要があるだけである。

もちろん、複数ユーザの価値はアプリケーションに依存する。 アプリケーションの複雑さと潜在的なユーザの数には相関関係があるように思われます。問題指向の言語にふさわしいアプリケーションは、多くのユーザが継続的に興味を持つ可能性があります。

さらに、一度基本的なプログラムが利用できれば、他の、たとえ関係のないアプリケーションを追加することも比較的簡単である。画面を読むことによって語彙をコントロールできるため、端末をさまざまな人が最小限の労力で使用することができます。また、各自が自分の好きな語彙を辞書に登録できる個人用画面を持つことができます。

このように、1つの端末からのメッセージ・トラフィックが十分に少なければ、必然的にそうなるのですが、事実上、コンピュータを人間の速度まで減速させているので、アクティブでないユーザをディスクに保存すれば、コアに収まるよりはるかに多くの端末、何百もの端末を処理することができます。

しかし、リエントラントプログラミングのルールが厳密に守られるようにするために、コストがかかります。ユーザ間でコンピュータの注意を切り替えるために必要な追加コードと、ディスクバッファとユーザ辞書に必要な追加コアは、シングルユーザアプリケーションを最初にデバッグすることを要求しています。そして、需要の高まりに応じて、複数ユーザの制御ルーチンでコンピュータの能力を倍増させる。マルチユーザコントローラに拘泥し、アプリケーションを完成させることができないのは、あまりにも簡単なことです。また、マルチユーザコントロールを完成させても、それを正当化するような需要を見つけることができない。

シングルユーザアプリケーションが成功したら、それをどのように多数のユーザに拡張できるかを紹介します。もしこのステップを踏むつもりなら、最初の実装で取るべき注意事項があります。 しかし、基本原則に留意してください!

### 7.0.1 ユーザなしの活動

各ユーザは、自分の状態を識別するためにレディテーブルのポジションを持つ。コンピュータはこのテーブルを調べて、次に何をすべきかを決定する。レディテーブルには、ユーザとは関係なく、コンピュータが実行しなければならないタスクを表す項目を追加したい場合があります。

例えば、電話回線にポーリングして入力を得る必要がある場合、他の作業とは非同期でポーリングを実行したいものです。割り込みルーチンは小さい方が良いので、文字セットの変換、パリティのチェック、メッセージの配信などのタスクは低い優先度で実行されるべきです。これは、レディテーブルのエントリを使用することで簡単に行えます。割り込みルーチンはメッセージルーチンを "レディ"に設定し、コンピュータは可能な限りそれを処理します。

このような独立したアクティビティは、それぞれレディテーブルエントリと、そのパラメータ；リターンアドレス、レジスタ内容などをユーザアクティビティと同じフォーマットで格納するための(おそらく)小さな辞書を持つべきである。 実際、これらのアクティビティは、ユーザを処理しないことを除けば、ユーザとほぼ同等である。この違いは重要なことで、エラーメッセージを生成しないので、何らかの方法で自分自身のエラーを処理しなければならないことを意味します。

まだお気づきでないようですが、私たちは今、オペレーティングシステムについて話しているのです。このテーマについてこれ以上話すことはありませんが、他にも必要な非同期アクティビティがあります。

- タイマーの割り込みを処理し、コアとディスクの時刻と日付を維持するためのクロック。一定時間、制御を放棄し他のアクティビティをレディにする機能を持つかもしれません。
- ディスクにブロックを書き込むルーチン。定期的にブロックバッファをスキャンして、コピーするブロックを探すかもしれない(しかし、読み出しルーチンがバッファを必要とするときにブロックを書き込む方が単純と考えます)。

このような活動はほとんどコストがかからず、通常、非同期の問題に対する最も単純な解答を提供します。しかし、基本原則に注意してください。

### 7.0.2 メッセージハンドリング

一人のユーザから入力を読み取ることができれば、多くのユーザから入力を読み取ることができる。 入力が利用可能であることと、それが誰からのものであるかを伝える割り込みを得る必要があります。あなたはそれを適切なメッセージバッファに導くだけでよいのです。 同様に出力もそうです。

単純である必要はありませんが、ハードウェアに排他的に依存することは確かです。 もし、端末をポーリングしなければならないのであれば、実に興味深いことになります。 しかし、この問題はこの本の範囲を超えていることに変わりはありません。

もし、すべてのユーザーをコア常駐にしない場合は、誰もコア常駐にしないほうがよいでしょう。 そうすれば、どんな入力メッセージもディスク上のメッセージバッファ領域に書き込むことができる。そして、すべての出力メッセージはディスクから読み込まれる。コアに常駐するユーザがいると不当に複雑になるし、ディスクアクセスはメッセージ送信に比べて高速なので、そのようなディスクアクセスを節約しようとするのは効率的ではありません。

## 7.1 ユーザの制御

複数のユーザがいるという事実は、新たな問題を生み出す。もちろんコンピュータは一度に一人のユーザしか処理できません (ここでは単一のプロセッサを想定しています)。しかし、あるユーザを処理し終えたら、別のユーザに注意を向けなければなりません。

1人のユーザに対する処理が終わるのはいつなのか? ユーザが入力待ちの状態であれば、コンピュータは終了していることは明らかです。ここではキーボード入力の話をしていますが、入力が完了するまでには何秒もかかります。同様に、ユーザが出力を送信している場合にも、同様にコンピュータは停止しています。特にデバイスからの応答が予想される場合は、出力に数秒かかります。出力中に処理を停止する必要はありません。あるメッセージを送信している間に、次のメッセージを作成することも可能です。しかし、そのようなオーバーラップを試みない方がずっと単純明解です。ユーザがディスクを読んでいるのであれば、コンピュータは停止することができる。

このような状況をカバーするために、一つのフレーズを定義したい。メッセージ送受信やディスクI/Oを実行するときはいつでも、ユーザはプロセッサの制御を放棄する、と言っておこう。これはユーザの自発的な行動であり、ユーザが制御を放棄するのはその時だけである。特に、ユーザから制御を奪うような時間単位(time quantum)は存在しない。これは以下のような理由からです。複数のユーザがいる場合、コードがリエントラントでなければならないことは明らかです。しかし、もしユーザが、自分が始めたことを最後までやり

遂げることが許され、自分が制御を放棄するとき以外は他の人に制御を奪われないと約束されていれば、リエントラントの要件はそれほど厳密なものではなくなります。プログラムはI/Oをまたいでリエントラントであればよいので、多くの手間を省くことができる。

では、ユーザがコントロールを放棄した場合はどうなるのでしょうか? コンピュータは単純にユーザのテーブルをスキャンして、他に準備のできた人がいないかどうかを調べます。このテーブルには、ユーザの辞書のアドレスと、readyかnotかのフラグが含まれています。I/O完了割り込みルーチンは、単に適切なユーザを準備完了とマークします。そして、誰も準備ができていない場合、コンピュータはテーブルを延々とスキャンします。当然ながら、プログラム起動時には誰も準備完了になっていない。

## 7.2 キュー(待ち行列)

ユーザコントローラにいくつかのコードを入れることで、手間を省くことができます。サブルーチンは2つ。QUEとUNQUEです。ユーザが、他の人が使っているかもしれない機能を必要とするとき、彼はQUEを呼び出します。もしそれが利用可能であれば、彼はそれを得る。もしそれが利用可能でなければ、彼はそれを待つ人々の列に加わる。それが解放され、自分の番が来れば、彼はそれを手に入れることができます。

例えば、他の人がディスクを読んでいたら、その人はディスクを読めません。あるいは、少なくとも特定のチャンネルやデバイスを使うことはできない。待っている間、もちろん彼はコントロールを放棄する。この機能を使い終わったら、UNQUEを呼び出して他の人に渡します。

これらは、非常に価値の高いルーチンです。各ディスク、各通信線(共有ライン)、プリンタ、ブロック1(ディスク割り当て)、非リエントラントルーチン(SQRT)など、この方法で処理できる設備はたくさんあるからだ。 拡張すれば、ブロックの排他的使用も可能になる。

当然、私はQUEやUNQUEを実装する具体的な方法をすでに理解している。そして、この場では、もっともらしい改造はうまくいかないことを、いつもより強く、私から注意しておきたい。その理由をすべて述べることにする。

ユーザの辞書アドレスとレディフラグに加えて、各ユーザはリンクフィールドを持たなければならない。そのユーザの辞書ではなく、ユーザコントロールの中に。保護されるべき各機能は、それに関連する2つのフィールドを持たなければならない：所有者と、最初に待機している待機人である。最適な配置は、各機能に1つずつ、このようなキュー用ワードのテーブルを持つことである。機能が空いている場合、その所有者は0であり、そうで

ない場合、その所有者はその機能を所有するユーザの番号である。もし誰も待っていなければ、その機能の待機人フィールドは0であり、そうでなければ、待っているユーザの番号である。

もし、その機能を使用したくなり、空いているなら、

- 自分の番号を所有者欄に記入して退出します。

もしその機能がビジーで、かつ、誰も待っていないなら

- 私は待機人フィールドに私の番号を格納し、0を私のリンクフィールドに置き、コントロールを放棄します。

誰かが待っている場合。

- 私は、待機人のリンクフィールドから始まるリンクの連鎖を、0リンクを見つけるまで辿り、そこに私の番号を置き、私のリンクフィールドを0にし、コントロールを放棄します。

機能を使い終わったとき(UNQUE)。

- 誰も待っていない場合、所有者フィールドを0にして退出します。
- 誰かが待っている場合、私は彼の番号を所有者フィールドに移動し、彼のリンクフィールドを待機人フィールドに移動し、彼をレディとマークし、終了します。

手順全体は単純で効率的です。多くの潜在的な問題を合理的かつ効果的に処理することができている。いくつかのコメントがあります。キューはおそらく非常に短いものでしょう。実際、コンピュータが過負荷にならない限り、機能は通常空き状態でしょう。一人のユーザが複数の待ち行列に参加することはできません。しかし、ユーザは複数の機能を所有することができます。 したがって、各機能に待機人フィールドが必要である：キューは各機能からたどらねばならず、各所有者からたどってはならない。この二つの概念は独立している。エラールーチンに、現在のユーザが所有する機能を解放するループを追加する必要があります。ユーザがキューに入るには自分の番号を知る必要があるので、この番号は彼の辞書のフィールドに格納され、再初期化ルーチンによって設定されなければならない。

これは複雑で面倒なことであり、複数のユーザを抱えることの代償でもある。

### 7.2.1 使い方

ブロックの排他的な使用を得るには、ブロック１を除いて、例外として扱うのが最善です。この目的のためにいくつかの機能キューワードを確保してください。空いているものを探し、それが表すブロック番号をどこかに保存しておき、そのブロックを他の機能と同様に扱います。最後の待機人がブロックを解放したとき、その機能キューワードを再利用のために解放する。 このテクニックはブロック自体には何の影響も与えないことに注意してください。ブロックはコアに常駐していても、いなくても構いません。誰でもそれを読んだり書いたりすることができます。しかし、他の誰もそれを独占的に使用することはできません。もしすべてのユーザが協力して、必要なときに排他的使用を要求すれば、完璧に機能します - 通常の読み書きに余分なコストはかかりません。実際、あるブロックの排他的利用が必要なのは、例外的な状況においてのみです。ブロック１がその一例です。このブロックは、他のブロックが読み込まれ、使用可能な領域が更新されるまでは、他の誰にも使われてはならないのです。

## 7.3 プライベート辞書

複数ユーザシステムに変換する場合の鍵は、あるユーザに関する必要な情報のすべてが、彼の辞書、つまりコアの単一の連続した領域に格納されることです。彼は、システムに属し、彼の辞書に存在しないコードを広範囲に使用する。一方、彼のアプリケーションに固有のコードは、そこに存在する可能性があります。ここで、最初の決断をしなければならない。ユーザのプライベート辞書には何が入っているのか?

コアの配置について見てみましょう。もし私たちがそう選択すれば、また、そう選択せねばなりませんが、それは辞書の形式に従っています：各エントリに続いて、それが実行するコードがあります。 各項目は、辞書を逆引きできるように、前の項目とリンクされています。スタックを制御するもの、辞書エントリを定義するもの、BASE や CONTEXT などのフィールドを指定するものなど、明らかにすべてのアプリケーションにとって関心のあるエントリがあります。他のエントリはおそらくローカルにかかわるものです。レコード内のフィールド名、テキストの編集に使用される定義、特殊用途のコード(乱数発生器、平方根など)。どこかの位置で、システム辞書とユーザ辞書を分離する必要がある。

複数のユーザ辞書を作成した場合、それぞれの最初の項目は、同じ時点でシステム辞書(図7.1)にリンクすることになる。したがって、各ユーザは他のユーザを意識することなく、辞書検索に影響を与えることはない。

### 7.3.1 メモリ保護

全ユーザが同時にコアに収まるなら、それで終わりです。メモリを適切な辞書に分割するのです。あるユーザが他のユーザに害を与えることがないように、メモリの保護を行う必要があります。先に述べたスタックと辞書のサイズチェックに加え、`=` 演算子のチェックを行い、ユーザが自分の辞書の外や、自分が読んだブロックの外に書き込めないようにする必要があります。ハードウェアメモリ保護機能がある場合、それを利用することは困難であることがわかります。 ユーザは、自分の辞書、システム辞書、ブロック・バッファを読むことができなければなりません。複数のユーザが同じブロック・バッファを書きたいかもしれません。同時にとは言わないまでも、少なくとも連続して書きたいでしょう。もしハードウェアが助けてくれるなら、それは私が見たどのものよりも良いものです。しかし、ソフトウェアの保護は十分なものにすることができます - 悪意のあるいたずらに対しては別ですが。

ユーザは他人を傷つけることはできませんが、自分自身を破壊することは可能です。したがって、彼の辞書を空に戻し、すべての制御フィールドをリセットするようなシステムエントリを用意する必要があります。このようなエントリは、やり直すには簡単な方法であるため、よく使われるでしょう。

固定サイズのエントリを実装している場合、システム辞書につながるリンクはない。すべてのユーザ辞書がシステム辞書と連続と限らないので、検索ルーチンは、ユーザ辞書とシステム辞書を別々に検索する必要があります。これは、わずかな命令数で済むはずですが、リンクされたエントリを好むもう1つの理由です。

辞書に複数の連鎖がある場合、それぞれの連鎖はユーザの辞書からシステム辞書にジャンプしなければならない。これは、辞書を再初期化するときだけの問題であり、システム辞書のチェーン・ヘッドのコピーを保持することで簡単に解決できる。

### 7.3.2 制御されたアクセス

冗長性を避けるために、システム辞書をできるだけ大きくしたいように見えます。しかし、必ずしもそうとは限りません。一部のユーザに対して特に拒否したい項目を除けば、システム辞書に入るかもしれない項目がいくつかある。その典型的な例が、ディスクの割り当てを制御するGETとDELETEのエントリである。無知なユーザがこれらのワードを誤って使用すると、ディスクに保存されているデータに大きなダメージを与える可能性があります。最良の解決策は、辞書エントリを使わずに、コードをシステム内に配置することです。このような性質のコードへのエントリポイントのテーブルを定義する。 そして、もしユーザがエントリポイントを使いたい場合は、まずそれを定義する必要があります。

- 17 ENTRY GET 18 ENTRY RELEASE

GETとRELEASEというワードを、テーブルの17番目と18番目の位置で識別されるコードで確立します。ライブラリのサブルーチン(FORTRANの算術サブルーチン)も同様に扱われるかもしれません。

ちなみに、これは一般的な保護方法を示しています。ワードを使うだけでなく、ユーザが正しく定義しなければなりません。明らかに、このプロセスをカスケードすることができます。悪意のあるいたずらに対するこのような保護の価値は、常に究極の保護である秘密保持に依存する。 しかし、機密保持がない場合でも、不注意による被害に対する貴重な保護となります。

# 7.4 ディスクバッファ

複数のユーザが同時にディスクを読み込んでいても、ディスクアクセスサブルーチンには全く影響がありません。ブロックバッファを検索して利用可能なバッファを見つけることができ、誰がそれを要求したかを気にする必要はありません。もちろん、少なくともユーザと同じ数のバッファを持つ必要があります。実際、辞書に必要でないコアはすべてブロックバッファに充て、アイドル状態にしておくのがよいでしょう。しかし、ブロックが読み込まれている場合、それが到着する前にそれがそこにあると仮定しないように、何らかの方法でバッファをビジー状態にマークする必要があります。ビジー状態のブロックを読もうとしたら、制御を放棄して、再スタートしたときに再挑戦すべきです。

# 7.5 ユーザの掃き出し(User swapping)

これまでは、すべてのユーザをコアに常駐させていました。これは、少数のユーザを扱うには、圧倒的に最適な配置です。同時に常駐できるユーザ数よりも多くのユーザを許可することは、哲学的には小さなことですが、実装は非常に難しい場合があります。例えば、コアに4人分のユーザ辞書を格納するスペースがあったとして、40人のユーザを許可したいとします。40人のユーザ辞書をすべてディスクに保存し、各ユーザがアクティブになったときにコアにロードすればよいことは明らかです。ディスクI/Oはメッセージ I/Oよりはるかに速いので、性能上のペナルティはありません。ユーザがメッセージI/Oを待っているとき、私たちはディスクに彼を書き出す。彼がメッセージI/Oを完了すると、我々はコアに彼を戻って読み取る。当然ながら、ディスクI/Oを待っているときにユーザをコアから追い出すことはしません。なぜなら、本来の遅延に比べて、書き出しと読み出しに不当に長い時間がかかるからです。

ここまでは問題ない。問題は、彼をどこに読み戻すかということです。もしユーザを常に同じバッファにロードするならば、4つのクラスのユーザが存在することになり、それぞれが1つのバッファに入ることができるのです。他のバッファが空であるにもかかわらず、1つのバッファで遅延が発生してしまうことを許さねばなりません。

どうせ苦労するなら、すべてのバッファを等価にして、空いているほうにユーザをロードすればよい。しかし、この場合、ユーザ辞書は再配置可能でなければなりません。つまり、ユーザ辞書への参照は、おそらくインデックスレジスタに格納されているその辞書の原点からの相対参照でなければなりません。最初から、つまりシングルユーザー・プログラムの時から、すべての辞書参照を相対的なものにすることを計画していたのであれば、これはそれほど悪いことではありません。しかし、そうでない場合は、プログラムをほとんど完全に書き直す必要があります。なぜなら、プログラム中に散在している辞書参照のすべてに対して、インデックスを付けなければならないからです。

実際、ブロックへの参照は、ブロックの(可変)原点からの相対参照でなければならないので、新しい問題が発生しているわけではなく、古い問題を拡張しているに過ぎません。しかし、もうひとつ複雑な問題があります。 システム辞書は絶対的なもの、ユーザ辞書は相対的なものというように、2つの辞書の間に本当の意味での区別ができるようになりました。したがって、同じ種類の項目でも、どちらの辞書にあるかによって扱いを変えなければならない。

たとえば、パラメータ領域にコンパイルされたコードがある場合、絶対的なユーザ辞書はアドレスフィールドにコードアドレスを格納することができます。しかし、相対的なユーザ辞書には、パラメータフィールドにジャンプするルーチンのアドレスを格納する必要があります。あるいは、相対アドレスは絶対アドレスと、おそらくはサイズによって区別され、適切に扱われなければならない。

解決不可能な困難を避けるために、シングルユーザプログラムは以下のような制約を設けて書くように注意する必要があります。

- ユーザ辞書の原点であるユーザポインタのためにインデックスレジスタを確保し、このインデックスを使用する。つまり、辞書は相対的なものとして扱う。
- すべてのコードをリエントラントにする。少なくとも、ユーザが制御を放棄する可能性のあるすべてのコード(ほとんどのコードであることが判明しています)を再入可能にします。

もし、多人数バージョンを実装するつもりが少しでもあるならば、このようにしてください。これは基本原則に反しますが、私たちは基本原則に反する価値があるほど基本的な問題を扱っているのです。

# 8. 思考するプログラム

意識の謎は、長い間、哲学者たちの興味を引いてきた。生命が複雑な組織の結果であるのと同様に、意識もまた複雑な組織の結果であることは、今や明白である。意識は、データ間の複雑な相互作用の副産物である。あまりに複雑な相互作用は、哺乳類の脳でしか起こりえない。

したがって、心を調べる方法の1つは、データを操作する実験をすることです。そのためには、コンピュータを使うのが一番だ。私たちは、これまで実現できなかった能力を持つプログラムを手に入れました。それを使って、「思考」の領域を探ってみてはどうだろう。私は、あなたが心理生物学者になることを提案しているのではありません。しかし、あなたのプログラムを単純に拡張するだけで、とても楽しい、そして本当に印象的なことができるのです。

ここでは、珍しい能力を持ついくつかのエントリについて説明します。もしあなたがそれらを使えるアプリケーションを持っていたり、それらを使うアプリケーションを作れるのであれば、是非とも試してみてください。しかし、基本原則では、目的もなくそれらを含めることは禁じられています。それらは十分に精巧で、十分に専門的であるため、予期しない状態で価値を発揮することはないのです。

私は、1つのプログラムの中に、今回説明するすべての項目を入れたことがあります。このプログラムは1500命令以下でしたから、すべてを一つのプログラムに含めることは現実的です。しかし、私は実験を続けていますが、いまのところ、その何分の一さえも必要とするアプリケーションを見つけることができませんでした。

## 8.1 ワードの解剖(Word dissection)

私たちのプログラムの最も厄介な特徴の一つは、ワードはスペースで区切らなければならないということです。句読点や演算子をスペースを介さずに接尾語としたい場合が非常に多いのです。そして、我々はすぐに接頭辞を望ましいものにする能力も追加する予定です。

スペース以外の文字を終端文字として認識するようにワードサブルーチンを修正するのは難しいことではありません。しかし、満足のいく一般性を提供することは不可能です。無数の特殊なケースを考慮することで、必然的にワードサブルーチンを過度に複雑にしてしまうのです。また、一般性を実現しようとすると、多くの工夫を無駄にすることになる。例えば、以下の文字列すべてがワードとなるような単純なルールは存在しない。

- HELLO GOOD-BY 3.14 I.B.M. -.5 1.E-3

同様に、以下の文字列を意図するワードに分離する単純なルールもない。

- -ALPHA 1+ ALPHA+BETA +X**-3 X,Y,Z; X.OR.Y

しかし、落ち込まないでほしい。これらのケースをすべて処理できる一般的なソリューションがあります。それは時間的に高価であり、おそらく非常に高価である。しかし、これより劣る解法はありえないことを示しながら、問題を徹底的に解決してくれるので、私はその価格に十分見合うものだと考えています。それに、テキストの処理速度は重要な要素ではありません。私たちは、このような贅沢をするためにこそ、速度を最大化するのです。

まだお気づきでないようですが スペースで終わるワードを読み、辞書を引き、数値に変換する。この定義でワードでない場合は、最後の文字を削除して再試行します。最終的には、残った文字がワードとなるように、十分な文字を削除する。

ここで、コストについて考えてみよう。落とすべき文字の数だけ、辞書検索( + 数値変換)を行う。これにより、高速検索と非数字の迅速な認識が可能になる。また、分解しなければならない文字列の長さを最小にすることも推奨される。しかし、現実的に考えて、分解が便利な場面は、値段に見合うほど多くはないだろう。 ただし、コンパイラのソースコードは例外だ。しかし、私はコンパイラを書くわけではないし、もしあなたがコンパイラを書くのであれば、ワードのサブルーチンを作ることで対応できるだろう。

注意しなければならないことがいくつかあります。アラインされたワードから文字をドロップするとき、このワード内の現在の位置を追跡する必要があります。しかし、次のワードを正しく開始できるように、入力ポインタをバックアップしておく必要があります。ちなみに、このためには、端末スペース上で最初のバックアップを行う必要があり、これは繰り返されません。

バッファリングされていない入力では、入力ポインタのバックアップは不可能です。第3章でバッファリングされていないデバイスをバッファリングするように提案したのはこのためです。解剖するつもりがないならば、基本原則を適用してください。

また、最後の文字を落としたことを検出する方法も必要です。カウンタを使うのも1つの解決策です。整列したワードのすぐ前にスペースを置き、そのスペースで止めるという方法もあります。辞書検索や数値変換の際にも使える便利なカウンタがないため、私は後者の方法をとっている。しかし、これでは、スペースを置く前に各文字をフェッチしなければなりません。これは、フェッチサブルーチンが逆方向の操作をしなければならないことを意味し、これまでで私が逆方向のフェッチを必要とすた唯一のケースです。これはハードウェアに依存します。

この解剖を洗練させるためにできることは2つあります。これらは互換性がなく、選択はあなたのアプリケーションによります。文字を一度に一文字ずつ落とす必要はありません。いくつかの文字が連続している場合、あるいはいくつかの数字がある場合、あるいはおそらくその組み合わせの場合、すべてをドロップしてから1回の検索・変換を行うかもしれません。この場合、1文字ずつ調べる必要があります(上記の2番目の終了を意味します)。また、英数字と特殊文字を区別する必要がある。そのためには、特定の文字セットとアプリケーションに合わせた64文字の文字タイプのテーブルが必要です。ハードウェアが許せば、64ビットのテーブルを使うことができるかもしれません。このあたりは古典的な時間と空間のトレードオフですね。

しかし、これは文字列を分解できないことを意味します。一方、分解したい場合もあります。例えば、複数形は終端の's'を削除することで簡単に格納できます。一方、文字列を分解してしまうと、簡単にワードを誤認してしまう。以前、SWAPというワードを分解してみたことがある。Sは定義され、Wは定義され、私のエラーメッセージはAP？一文字落とすときは、語幹を示すダッシュで置き換えるべきかもしれません。あるいは、識別不能なワードは誤認識でもかまわないのかもしれない。

さらにもう一つ注意があります。解剖を行う場合、最初にワードを揃えるときに余分な文字を捨ててはいけません。入力ポインタは、正しくバックスペースできるように配置しなければなりません。もし、最大ワードサイズを超えたら、直ちに停止し、ターミナルスペースを供給してください。つまり、1つのワードが最大サイズを超えることはなく、現在では最大文字列サイズになっています。

もう一つの最適化は、辞書のワードのサイズに関係する。 ワードの一部にしかマッチしない場合、数値形式が許すなら、その時点で文字を削除し始めた方がよいでしょう。

プログラムにとって、ワードの分解はどのような意味を持つのでしょうか。プログラムが「考える」ことにどのように役立つのでしょうか? それは、プログラムがあなたの心を読むことができるということです。つまり、あなたがどのように何かをタイプしても、コンピュータはあなたが意図した意味を取り出してくれるということです。コンピュータは、左から右へのスキャンで、できる限り長い文字列の意味を使います。例えば、「+1」を定義した後に「+1000」と入力すると、誤認識してしまいます。しかし、一貫性のあるワードを使えば、それに従うことになります。

私は、この能力が人々に感銘を与えるものであると言いたいのです。少なくともあなたは感動するはずです。しかし、あなたの上司のような普通の人は、コンピュータにこのような能力を期待しているのです。それがないことを発見したとき、彼らは否定的な印象を受けるだけなのです。

# 8.2 レベルの定義

これから説明することの標準的な用語を知らないのは残念なことです。議論されているのを聞いたことがないし、 検索したこともない。しかし、この用語はコンパイラを書く際の標準的なものであり、コンパイラを扱う講座で取り上げられるものであるはずです。この用語を知っている人は、これから話すこともほとんど知っているはずです。その応用を広げてもらえるといいのですが。

私たちの算術演算子はすでにスタック上に引数を発見している。従来の代数表記法では、このような演算子は中置演算子(infixes)として使用され、左から右へのスキャンでは演算子が発見されたときにオペランド 1 つだけしか提供されません。その結果、演算はもう一方のオペランドが利用可能になるまで延期されなければなりません。

さらに、その他の演算子が使用可能になるタイミングを制御するよりも、操作の階層を設けています。例えば

- A+B*C

の場合、乗算は加算の前に行われなければならない。さらに、括弧は標準的な階層を修正するために使用されます。

- A*(B+C)

このような表記法は、我々の表記法と全く同じである。「オペランドを-演算子の-前に置く」に対する利点はなく、いくつかの制限もある。しかし、人々はそれに慣れ、それがないことに否定的な印象を持つのです。そこで、その機能を提供する方法を紹介します。

しかし、私たちの注意を通常の算術演算子や論理演算子に限定する理由はない。他の似たような階層をいくつかお見せしましょう。私が説明する機能は、それらすべてを扱うことができる。

新しい種類の辞書エントリを作成しよう。これは、レベル番号という数字が付加されていることを除けば、定義と同じものである。そこで、これをレベル定義と呼ぶことにする。ルールとしては、レベル定義は遭遇したときに実行されるのではなく、プッシュダウンスタックに置かれる。同じかそれより小さいレベル番号の別の定義に出会ったときに実行される。

レベル定義の便利な書式は次の通りです。

- 2 :L word ... ;

2はスタックから取り出されたレベル番号である。`:L` はワードをレベル定義付きで宣言します。';' は終わりを表します。

では、+と*について説明します。

- 0 :L , ;
- 1 :L + + ;
- 2 :L * * ;

これらを古い定義から、レベルの定義として再定義しました。また、','は停止するための方法として定義しました。これで、次のように言えるようになった。

- 3 + 4 * 5 ,

何が起こったのでしょうか? 3 はパラメータスタックに、+ はレベルスタックに、4 はパラメータスタックに、 *はレベルスタックに (すでにある + よりも高いレベル番号を持つので)、5 はパラメータスタックに入ります。ここで、','は* を強制的に実行し (レベル番号が小さいため)、 * パラメータスタックに 5 と 4 を見つけます。',' はまた、(引数 20 と 3 で) + を強制的に実行し、そのレベル番号が 0 であるため、それ自身は実行されず何もしない。

お分かりいただけたでしょうか? 読者がこのテクニックに詳しくなってほしいと思いたいところですが、あえてそうする必要はないでしょう。私が本当に貢献したいのは、通常コンパイラに組み込まれている技術を、辞書の項目を使って実装する方法だけです。そのために、算術演算子を定義し、自分で例を作ってみることをお勧めします。同じレベルの演算子はお互いを強制的に排除し、より低いレベルの演算子はより高いレベルの演算子を強制的に排除することを忘れないでください。演算子の相対的なレベルを間違って推論してしまうのは、不思議なほど簡単なことです。

ここまでで、何がわかったでしょうか。なぜ、レベルの定義に興味があるのでしょうか? レベルの定義は単純です。レベル定義は普通の定義に比べて単純になりがちです。しかし、レベル定義があれば、どんな言語でもコンパイラを書くことができるのです。 レベル定義は、現代の言語の基本であるLR-1文法だけでなく、あらゆる文脈自由文法を実装するのに必要かつ十分なものです。正直なところ、レベル定義が提供する力をどう使えばいいのかわからないが、後でいくつか提案してみよう。

さて、仕事に戻りましょう。レベルの定義をいくつか見てきましたね。少しは遊んでいただけたでしょうか。どのように実装するのでしょうか? そうではありません。 そうではなく、一般化したもの、つまりレベルエントリを実装します。レベルエントリの用途を見つけたとき、レベル定義をそのまま実装する方が、私がやっていた方法よりも安上がりであることもわかりました。

第5章で説明したように、すべての辞書エントリは仮想的なコンピュータの命令と考えることができる。レベルエントリは、レベル定義の流儀に倣って、その実行を遅らせることができる命令と考える。なぜそうしないのか? 定義とは、結局のところ、ある種の命令に過ぎない。もし、それが有益に遅延できるのであれば、他の命令もそうかもしれない。

複雑に見えたらごめんなさい。そうなんです。あなたは何もしないで何かを得られるわけではありません。でも、それだけの価値があるのです。しかし、今やっていることはすべて、これまでやってきたことの上に積み上げることに注意してください。特別な種類のエントリという概念は、辞書が利用できるかどうかにかかっており、レベル番号を含む定義の拡張は、定義があるかどうかにかかっていることに注意してください。私たちは徐々にツリーを構築しており、今や高い枝にいる。下位の枝のすべてに依存することはないかもしれませんが、いくつかは持っていなければなりません。

レベルエントリはどのように実行するのですか? 他のものと全く同じです。 ただし、レベルエントリが最初に行うことは、レベル番号をパラメータとしてLEVELルーチンを実行し、名前を付けることです。LEVEL はこのレベル番号をレベルスタックに対してテストします。3つのケースが発生します。

- レベル番号とエントリをレベルスタックに置き(上位エントリ)、RETURNする。
- レベルスタックの先頭をこのエントリで置換し、古い先頭を実行することができる。
- レベルスタックが空で、レベルが0の場合、このエントリを実行する。

3つのケースはすべて必要です

実際にスタックからエントリを実行する前に、LEVEL は別のルーチン FORCE を参照するために SOURCE アドレスを設定する必要があります。メイン制御ループは、ワードを読み込んで検索するか、定義からフェッチすることで、次のエントリを取得することを思い出してください。ここでは、3つ目のソースとして、レベルスタックを使用します。定義に関しては、SOURCEと仮想ICの古い値は、リターンスタックに保存されなければならない。

最終的にレベルエントリを強制的に実行するとき、それがすでに実行されたことを思い出し、すぐにLEVELにジャンプしなければなりません。 この再実行は、ルーチンのアドレスの 1、2 命令下の別の場所から開始する必要があります。あるいは、再実行開始アドレスをパラメータとして含み、レベルスタックに保持することもできます。

レベルエントリが終了すると、RETURN し、制御ループはFORCE に移行します。FORCE に行くには、レベルエントリを完了させるしかありません。その機能は、レベルスタックをチェックし、他のエントリが上にあるエントリによって強制終了させられるかどうかを確認することです。3 つのケースが発生します。

- レベルスタックを放置し(上位のレベル)、SOURCE と virtual-IC を return-stack から復元し、RETURN する場合。
- 下位のエントリを実行し、上位のエントリと入れ替え、レベルスタックを削除する。
- 下位のエントリがなく、レベルが 0 の場合、先頭のエントリを実行し、レベルスタックを空にします。このとき、リターンスタックから復元する。

ここで、リターンスタックの重要性とSOURCE を保存する必要性を強調しておこう。レベルエントリが実際のところ定義である場合、SOURCE は再びリセットされます。戻ってFORCE に再び遭遇するまでには、しばらく時間がかかるかもしれません。実際、レベル定義は定義内で発生することもあり、他の定義を実行することもある。全体のプロセスは理解できないほど複雑に絡み合っており、実際にそうなっています。しかし、それは自ずと解決されるでしょう。定義、特にレベル定義の美しさは、再帰的関数の美しさに似ています。定義を行う際には、単純なケースだけを考慮すればよく、複雑なケースはそれ自体で解決されます。

これで、レベルエントリとその定義を実装することができるようになりました。これで何ができるようになったか?

- 一般的な算術演算子: + - * / MOD ** を定義することができます。
- 論理演算子: OR AND NOT-IMPL を定義することができます。
- 中置の関係演算子: = < > <= >= /= を定義することができます。
- 中置の代入演算子: = := (どちらかの方向に作用するもの)を定義することができます。
- 上記のすべてを定義することができます。

以下のように定義はアプリケーションに依存します。

- PLUS MINUS TIMES DIVIDED-BY EQUALS のようなワード: 英語算術式を定義することができます。
- MOVE...TO... や DIVIDE...INTO...や ADD...TO...のようなフレーズ: COBOL 言語風算術式を定義することができます。

しかし、2 つの特別な使用法を挙げよう。

- 次のような形式の文を考えてみよう。 o IF relation THEN statement ELSE statement ;

IF を定義して、それが THEN によって強制終了されるようにし、条件分岐を生成する。THEN は ELSE によって強制終了され、IF によってぶら下がったままのアドレスを修正するように定義する。ELSE を定義すると、まず無条件に分岐し、THEN を強制終了させ、その後自分自身が強制終了されるのを待ちます。ELSE を強制終了させ、前方分岐を修正するために ; を定義します。

このようなコンパイラの構成は、いくつかの記述で実装することができます。

- 次のような文を考えてみよう。

o 1800. FT / SEC ** 2

定数を即座にスタックに置き、強制的に乗算のように動作するエントリ UNIT の一種を定義してください。1 を即座にスタックに置き、強制的に割り算をするように `/` を定義します。**を中置演算子として定義し、FT と SEC を UNIT として定義します。

この式は正しく評価されます。とともに、あなたがどのような式を作ったとしても正しく評価されます。

私はあなたにボールを渡します。自然言語入力フォーマットから利益を得ることができるアプリケーションがあなたにあれば、それを実装するためのレベル定義が可能です。例えば、高校の物理の教科書の巻末にある問題を解くようにプログラムを教えるのは難しいことではないでしょう。

レベルエントリはコンピュータの能力を高めるものではないことに留意してください。単に、コンピュータにとって不自然な順序で命令を指定できるようにするだけです。まずアプリケーションを動作させることが大事です。派手な制御言語を追加するのはそのあとで十分です。

このことは、プログラムが「考える」こととどのように関係しているのだろうか? ただ単に、制御言語を人間指向の形式に委ねるだけです。私たち以外には、これさえも印象に残らないのです! そして、FORTRAN の式評価器にはもはやどれほどの感銘を受けているのでしょうか。

# 8.3 無限の辞書

あなたの辞書は平均して数百の項目があると思います。少量のデータでも1つのソースを紹介するためだけでも、多くのフィールドが生成されるようです しかし、アプリケーションによっては、より大きな語彙を必要とするものもあります。例えば、1万人の顧客のうちの1人を特定する必要があったり、104種類の元素記号が必要だったり、1000種類の認可済み添加物の名称が必要だったりする。

このようなボリュームは、明らかにディスクに保存しなければならない。また、ディスクの検索を明示的に行わねばならないのも困りものである。ありがたくも効果的な解決策がある。コア辞書でワードが見つからず、かつ数字でもない場合は、ディスク上のブロックを検索するのだ。さて、問題は次のようになる。どのブロックを検索するのか?

CONTEXTと呼ばれるフィールドを作成します。これは、ブロックを特定すると同時に、それがコアのどこにあるのかを示唆するもので、ブロックアドレスと同じように扱います。このブロックを検索してください。CONTEXTを変更することで、異なるディスク辞書を検索することができます。複数のブロックをリンクさせることで、より大きなディスクを検索したり、複数の辞書を順番に検索したりすることができる。

ワードが見つからなければエラーメッセージを生成することになるので、かなりの量のディスクを検索する余裕があります。そのワードが見つからないことを確認するためにメッセージの印字を遅らせることは、素晴らしい投資です。しかし、何千もの項目があるような大規模な辞書の場合、このような方法では適切といえません。

非常に大きな辞書の場合、ワードをごちゃまぜにして(scramble)ブロックアドレスにし、そのブロックを検索します。これは、コア辞書のマルチチェーンで行ったように、ワードの文字からブロック・アドレスを計算することを意味します(おそらく別のアルゴリズムが必要でしょう)。そのワードがどこかにあれば、1000のブロックのうち1つを検索すれば、そのブロックの中にあることが保証されます。なぜなら、ワードを見つけるのと同じスクランブル技術を使用して、そのワードをそこに置くことができるからです。多くのワードが同じブロックにスクランブルされるので、もちろん完全に一致するものを探します。これもコアと同じです。このように大きなディスクの辞書では、いくつかのことに気をつけたい。第一に、一度選んだスクランブル・アルゴリズムは二度と変更できないので、多くのエントリを定義する前に良い選択をすることである。第二に、すべてのブロックのエントリ数をほぼ同じにし、ブロックの容量の半分にほぼ等しくすることである。あるいは、ブロック同士をリンクしてオーバーフローを防ぐ。

このようなディスク辞書は、コンピュータに詳しくない人にとっても、非常に印象的なものです。なぜなら、膨大な語彙に高速にアクセスできるからです。高速とは、1回のディスクアクセスで、何万もの項目を検索できることを意味する。

ディスク辞書の項目はどのようなものでしょうか。私は、2つのフィールドで十分であることを発見しました。それは、コア辞書のワードフィールドと同じ大きさのワードフィールドと、1ワードの長さのパラメータフィールドです。ディスク上で一致するものがあれば、そのパラメータをスタックに置くのです。ディスクに絶対アドレスを保存する余裕はないので、コアのようにアドレス・フィールドを持つことはできないことを忘れないでください。コード化されたアドレス・フィールドを用意することもできますが、ディスク・エントリを定数として扱うのが適切だと思われます。

例えば、ブロックに名前を付けることができます。ブロックの名前を入力すると、そのアドレスがパラメータフィールドからスタックに移動されます。 なぜなら、ブロック番号そのものを入力すると、そのアドレスはスタックに置かれるからです。ブロック番号とブロック名は、同じように使うことができます。したがって、アカウント番号を入力すると、そのアカウントに関連するブロックがスタックに配置され、そのフィールドが参照するベースワードに格納されます。不正なアカウントは、通常の方法で、エラーメッセージを引き起こします。あるいは、あなたのコンピュータの命令に名前を付けることもできます。そして、その名前を入力すると、1 ワードの命令がスタックに置かれ、次の処理に備えることができます。

アカウント番号の話をしましたが、ブロックに番号を付けることはできないことに注意してください。つまり、ディスク辞書のエントリ名を数字にすることはできない。なぜなら、数字を入力するとスタック上で変換され、ディスク上で検索されないからである。また、ディスクを検索する前に変換を試みないと、文字を入力するたびにディスクを検索することになる。しかし、「数字」はNUMBER で定義される数字とあまり似ていないことが多い。ダッシュや文字などを埋め込んだり、文字の前に#をつけたりすることができる。

ディスクにエントリを置くには? 特殊な定義のエントリ:

- 0 NAME ZERO

を使います。これは、CONSTANT に類似したものです。あるいは、フラグを設定し、辞書エントリのサブルーチンにディスクとコアのどちらを使うか決定させることもできる。 ディスクとコアのどちらかがに配置される可能性のある異なる種類のエントリがある場合、後者の方が望ましい。

また、ディスクエントリを忘れるための方法も必要である。

- FORGET ZERO

FORGETは、ZEROというワードの典型的な使い方ではないため、エントリを定義するときのようにWORDを呼び出す必要があります。エントリを見つけると、パックしようとせず、そのエントリを単純にクリアします。エントリルーチンは、まずディスクを検索して、そのワードがすでにそこにあるかどうかを確認する必要があります。コアでは有用であったかもしれませんが、ディスクに複数の定義があることは好ましくありません。それから、穴を探します。もしそのワードがすでに存在するか、あるいは穴が見つからなかったら?エラーメッセージが表示されます。

リファインの話をしよう。ディスク上に1000の名前があると、ニーモニックが不足しがちです。CONTEXTというフィールドを再利用してみよう。ワードをブロックアドレスにスクランブルした後、CONTEXTの内容を追加してそのブロックを検索する。CONTEXTが0であれば、違いはありません。しかし、CONTEXTが0以外であれば、別のブロックを検索することになる。CONTEXTが0から15まで変化することができれば、同じワードでも16種類の定義があることになります。定義したときにCONTEXTの値が同じだったものを探すことになります。あるCONTEXTの下にそのワードのエントリがない場合、マッチは得られません。異なるCONTEXTの下で同じワードの定義を含むブロックは、検索されません。

例えば、在庫番号は、異なるセールス・ラインでは同じに見えるかもしれません。CONTEXTを設定することによって、それらを区別することができます。レポート画面とその指示画面は同じ名前でよいので、CONTEXTで区別してください。どうせスクランブルをかけるなら、CONTEXT(2の累乗)を入れたほうがよいでしょう。実際、CONTEXTはこれまで説明した2つの方法で同時に使用することができます。なぜなら、付加定数としては小さく、ブロック番号としては大きくなる傾向があるからです。従って、検索ルーチンは、そのサイズに基づいて、スクランブルをかけるかどうかを決めることができます。

しかし、ワードの解析を行う場合、問題が発生する。辞書の検索に加え、ディスク検索とそれに伴うディスクアクセスが発生するのである。いくつかの解決策が考えられる。最初の数文字だけをスクランブルし、少なくともディスク検索は同じブロック(コア)になるようにする。あるいは、CONTEXTに0以外の値のみを使用し、0がディスク検索しないようにする。つまり、分解とディスク検索を相互に排他的にするのです。よくあることですが、この問題を意識していない場合、深刻な問題となります。

# 8.4 無限のメモリ

もちろん、本当に無限のメモリを持つことはできません。無制限のメモリでさえも。しかし、コンピュータが利用できる全てのランダムメモリに直接アクセスすることは可能です。第4章で紹介したフィールドエントリを少し拡張すれば、それが可能です。出力とは特に関係がなく、「思考」に関係することで十分印象的なので、ここまで議論を先延ばしにした。

無限のメモリをどうするかという問題は、あなたにお任せします。どうにかして整理しなければならない。さまざまな部分を調べたり、フィールドを移動させたり、好きなように。私ができるのは、ディスクへの明示的な参照を排除する方法を示すことだけです。

ディスクのアドレスを指すパラメータをフィールドに含めることにしましょう。 フィールドはそのアドレスからの相対的なもの、つまりそのディスクブロックに含まれることになります。プログラムは自動的にそのブロックを読み込んで、フィールドを取得します。もちろん、いくつものフィールドが同じブロック・アドレスを指すことになります。

異論を唱える前に、話を先に進めておこう。ブロックアドレスと一緒に保存されているのは、そのブロックが最後に使用されたコアバッファの位置です。 したがって、ブロックがオーバーレイされない限り、プログラムは実際にディスクを読む必要はなく、ブロックのためにコアバッファを検索する必要もありません。そのため、同じブロックに繰り返しアクセスしても、ほとんどコストはかかりません。

いくつかのトレードオフがあります。オーバーレイを最小にするためには、コア・バッファの数に余裕を持たせる必要があります。このような使い方を念頭に置いて、ブロックサイズを選択する必要があります。しかし、ディスク上に散らばったデータを簡単にアドレス指定できることと、あるデータはディスク上にあり、他のデータはコアにあるということを忘れることができる素晴らしさは、スピードの低下を補って余りあるものだと私は思います。それはさておき、その機能を効率的に実装するのはあなたの問題です。

例えば、ディスクの一部をスキャンするとする。フィールドを定義し、そこでループを確立するだけをすればよいです。最初のブロックアドレスから始めて、それをフィールドが期待するベース・ロケーションに格納し、ループを通過するたびにそれをインクリメントする、というループです。さて、この利点はわずかなものです。節約できるのは読み出し命令だけです。しかし、そのブロックが別のブロックにリンクしている場合、他のフィールドのベースロケーションにリンクを格納するだけでよく、リンクが関与していることを

忘れることができます。リンク先のフィールドにアクセスすれば、自動的に読み込まれます。そうでない場合は、読み込まれません。データが複雑であればあるほど、そのメリットは大きくなります。

もちろん、ブロックの書き方も気にする必要はありません。 第6章では、ブロックをすぐに書き込むのではなく、書き込みが必要なブロックにフラグを立てることについてお話しました。これが功を奏します。フィールドを変更すれば、そのブロックは書き直され、変更しなければ書き直されないのです。ただ、GOOD-BY と言ったときに、あなたのプログラムが変更されたブロックを確実にすべて書き込むことを確認してください。

ベースアドレスへのポインタを 0 にすることで、コアにアクセスするフィールドエントリと同一にすることができます。ディスクアドレスを指さないなら、コアを指すと意味しなければならない。

フィールドエントリにベースを追加することで、COBOL のデータ分割におけるレベルと非常によく似たデータ構造が定義されることに注意してください。01 レベルはディスクアクセスに対応する。02 レベルはフィールドそれ自体を指す。 いくつかの命令を追加すると、より高いレベルを追加することができる。ポインタがディスクアドレスではなく、他のフィールドの記述を参照する場合、03 レベルと同等のものがある、など。

フィールド参照が実際にどのように機能するかを考えてみましょう。フィールドのエントリには、そのフィールドがどのワードにあるか (あるいはどのワードから始まるか) を示すワードパラメータがあります。このフィールドが他のフィールドを参照している場合、ワード・パラメータを足し合わせる。ディスクブロックのコアアドレスを見つけると、ワードオフセットを追加して、ほら、目的のワードを手に入れた。中間フィールドが変更されない限り、中間フィールドを経由する利点はほとんどない。なぜか? ベースフィールドアドレスをインクリメントすることで、マトリックスの異なる行やブロックの異なるレコードにアクセスすることができる。また、1つのレコードの異なるサブレコードにアクセスすることもできます。とても便利です。COBOL はかなりいい言語だと思わせるに十分だ。もちろん、コア上のフィールドでも同じことができる。ただ、結局のところディスクアドレスを指すことはない。

一言、警告しておきたい。ブロックのコアアドレスをインデックスレジスタに入れることで効率を上げようとしないでください。インデックスが現在どのブロックを識別しているのか、追跡するのが大変だからです。有用な一般性を提供するためには、かなりの量のコードを通さなくてはなりません。もちろん、ハードウェアには特別な機能があるかもしれません。マイクロプログラミングとか? 間接的なアドレッシングも有効かもしれません。

このような精巧なアドレッシング機能があれば、画面のデバッグに役立てることができます。メモリ保護は簡単で、とても役に立ちます。各フィールドのエントリに、そのフィールドの最大サイズ(ワード単位)を含めます。そのフィールドにあると称するアドレスを計算するとき、それがそうであることを確認する。最終的なブロックリファレンスの上限は、もちろんブロックサイズです。コアリファレンスの上限もわかっている。OVERFLOWというシンプルなエラーメッセージは、トラブルが伝播する前にキャッチすることができる。

フィールドエントリの種類を追加して実装したくなるかもしれません。このエントリはリンクを持っている。もしあなたがフィールドの外にある参照をした場合、このリンクをたどって要求を満たそうとする。特に、ブロックを指すレコードエントリ。ブロックの終わりを越えてレコードオフセットをインクリメントすると、ブロックからリンクを拾い、ブロックアドレスを変更し、レコードオフセットをリセットし、新しい、オーバーフローしたブロックにアクセスすることができます。それも自動で!? これは、レコードが固定長の断片で構成されている場合、可変長のレコード(実際にはブロック)の非常に魅力的な実装になります。

このようなオーバーフロー機能が必要な場合、リンクを構築する方法を提供しなければならない。レコードサイズの穴をブロック(チェーン)から探すエントリが必要である。もちろん、すべての穴は同じサイズである。もし穴が見つからない場合は、新しいブロックをGETし、それをリンクしなければなりません。その結果として丸ごと穴のブロックが一つ得られます。穴は最初のワード、文字、ビットに0を入れることで識別され、すべてのレコード位置が空になるように、GETで新しいブロックを0にクリアする必要があります。当然ながら、オーバーフローしたブロックが互いに近接している保証はありません。ほぼ間違いなく、そうではないでしょう。そこは気にしないか、最初に各ブロック・チェーンを平均サイズまで順次割り当てるか、どちらかです。

レコードを削除するのは簡単です。最初のワードに0を格納することでホールを作る。これによって、連鎖を解くことができる空のブロックが発生したかどうかを発見するのは難しい。データの縮小が予想され、スペースを回復する必要がない限り、悩む必要はない。データはどのように縮小するのでしょうか? また、レコードを移動させることもしないでください。ブロックアドレスを絶対化するのと同じように、レコードアドレスも絶対化して(もしレコード・アドレスを使うのであれば)使いたいのですから。

こうして、ディスク上に散在するフィールドに自動的にアクセスすることができ、しかも可変長のレコードにアクセスすることができる。基本原則!

1つだけ! フィールドエントリが他のフィールドエントリをアドレス指定できる場合、フィールドとディスクアドレスを区別する何らかの方法が必要である。これに関して私は何も提案していない。

# 9. 立ち上げのプログラム

ものごとがどのように始まったのかを理解することは、時に難しいものです。私たちはこれまで、あなたのコンピュータにはコンパイラがあり、あなたがそれを使ってプログラムをコンパイルするものと暗黙のうちに考えてきました。しかし、そのコンパイラはどのようにして作られたのでしょうか? 今日、その答えは間違いなく、別のコンピュータで別のコンパイラによって作成されたものである。私たちは、「すべての生命は、以前に存在した生命から生まれる」という生物学的な格言と同等なことを達成したのです。実用的には、すべてのプログラムは以前から存在したプログラムによって準備されているのです。

このことは、コンパイラを書く人にとって、特にターゲットとなるコンピュータがまだ作られていない場合には、いくらか楽なことではあるが、欠点もある。 既存のプログラムへの最終的な依存は決して捨てられないのだ。ある命令を生成するコンパイラを使ったり、あるディスクフォーマットを前提としたりすると、互換性を確保するための制約を受けることになります。 私たちのプログラムの単純なバージョンは、コンパイラの動詞を含んでいれば、それ自体をコンパイルすることが完全に可能であることを考慮してください。これは標準のコンパイラよりも自由度が高いのですが、より重要なのは、標準のコンパイラを捨てることができるということです。

第1章では、ソフトウェアの品質に関する悲しい現状を説明した。優れたオブジェクトプログラムを用意できても、コンパイラという不幸な妥協の産物のために、ソースプログラムとして維持しなければならないのである。これがプログラムを始めるのに最も便宜的な方法であることは認めざるを得ない。しかし、再コンパイルや修正という長い時間をかけても最も効率的であるかどうかは疑問です。

自分のコンピュータにアクセスできる状況を想像してみよう。 つまり、一人のユーザが、一度に何時間かぶっ通しで、すべての照明がついたボードの前に座っている状態です。これは確かに典型的な状況ですが、あなたが有能で、一生懸命に働き、変則的な時間でも働くことを良しとするならば、いつでも手配することができます。あなたとコンピュータはプログラムを書けますか? 既存のプログラムから派生したのではないプログラムを書けますか? 少しは勉強になるし、やってみるととても楽しいよ。

# 9.1 スタートを切る(Getting started)

コンピュータの電源の入れ方、切り方、コンソールスイッチからのデータの入力と表示、ディスクに保存されているデータの破損を防ぐ方法など、まず知っておかなければならないことがある。そのような情報は、オペレータやエンジニアを追い詰めなければ発見できないかもしれない、人の注意をひくことがまれなので、書き留められることはないのだ。

さて、あなたは今、コンピュータと向き合っている。どうするんだ? まず練習です。コンピュータを起動したら、無限ループを実行するように割り込み位置を初期化しなさい。いいかい? それからループを修正してメモリをクリアするようにします。いいですか? おそらく多くのことを学んだことでしょう。

さて、これからが本番です。まだ使うことはできないにしても、辞書の構築を開始します。ここでエントリの形式を選びます。可変長エントリは必須ですが、それでも、ワードサイズとレイアウトはあなたが決めることができます。最初のエントリはSAVEで、これはプログラムをディスクに保存します。コントロールループがないので、手動でジャンプしなければなりませんが、少なくとも、多くの作業をやり直すことは最小限に抑えられます。2番目の項目はLOADで、ディスクからプログラムを再ロードします。ハードウェアのロードボタンがあるかもしれませんが、それと互換性を持ってプログラムを保存できるのであれば、それはそれで結構です。そうでなければ、ロードカードにパンチして初期ロードを提供するのがよいでしょう。しかし、コアから再スタートできるのは常に便利なことです。

3番目のエントリはDUMPで、これはコアをプリンタにダンプします。スイッチで見るよりずっと速いので、それほど速くなくてもよいでしょう。このルーチンはおそらく自明なものではありませんが、12命令以上かかることはないはずです。ほんの少し延期してもいいかもしれません。

さて、数時間の作業で、(マニュアルを先に読んでいれば)あなたはオペレーティングシステム(SAVE, LOAD)とデバッグパッケージ(DUMP)を手に入れることができました。そして、あなたは自分のコンピュータについて多くのことを知っているのです。

# 9.2 根をつくる(The roots)

心配しないでください、私はこのプロセスを自分で経験しました。実は2回やっているのですが、自分がやったようにではなく、今思うと、こうすればよかったと思うようなことを書いています。つまり、アドリブの余地があるということですね。

ある意味、私たちは一本の木を作っているようなものです。今、根を作り始める段階にきています。しばらくは何もかもが隠されていますが、やがて日光に到達すれば枝を伸ばし始めるでしょう。

プログラムを LOAD し、コアを DUMP することができますね。そろそろスイッチから離れ、タイプライタを使うべき時です。メッセージバッファをセットアップして、そこからテキストを送受信できるようにしましょう。 おそらく、テキスト入力を待っている間、あなたのプログラムはどこかで無限ループに陥っているはずです。そのループを認識できるようになりましょう。実行時間のほとんどをそこで過ごすことになるのですから、すべてがうまくいっていることを知ることは心強いことです。

メッセージ I/O に関連する辞書項目はありません。定義することもできますが、私たちはそれを必要としません。一般に、エントリは必要なときだけ作成します。必要なエントリは後でいつでも追加できます。

うまくいきましたね。スタック、辞書検索サブルーチン、WORD と NUMBER のエントリを作成しましょう。ここは注意を集中して正しいエントリを作成してください。つまり、NUMBER を単純化して、後でやり直してもいいやなどと思わないでください。スイッチを使ってもやり直しの方が総作業量は多くなります。

今度は制御ループを書きましょう。スタックをテストするかもしれませんが、不特定エラーのルーチンにジャンプしてください。そして実行します。DUMP はまだ唯一の出力ルーチンですが、DUMP, SAVE, LOAD といったワードを読んで実行できるようにしてください。

ENTRY のエントリを書きます。これは辞書の項目を構築するサブルーチンです。WORD、NUMBER、ENTRY については、実行されるコードを指定していません。これらはサブルーチンであり、その名前を使うのはコードをコンパイルするときだけです。ですから、これらは呼び出し命令を生成するコードを実行する必要があります。まだ、そのコードは書いていません。

次に、コードエントリと呼ばれるコードに名前を付けるワード、そしてデポジットと呼ばれるスタックをコアに置くワードを定義します。これで 8 進数を入力して、辞書に格納できるようになりました。もうスイッチは必要ありません。また、コード用に新しい辞書エントリを作成することもできます。

# 9.3 枝をつくる(The branches)

私たちは一つのマイルストーンに到達しました。目に見えない作業は終わり、残ったものを文書で記録することができるようになりました。やるべきことはたくさんあり、その順番はそれほど明白ではありません。私たちはソース言語を手に入れたので、それを修正し、すべてをやり直すことなく再コンパイルできるようにする必要があります。陳腐化する一時的なコードを生成せざるを得ないが、その分、労力は軽減される。

まず、READ とWRITE のエントリで、1 つのコアバッファにディスクアクセスを提供します。次に、そのブロック内のテキストを入力したり、置き換えたりするためのシンプルなTとRです。これらのエントリは後で使わなくなるので、単純にしておきましょう。

次に、画面用の READ と ;S の動詞が必要です。ブロック番号を指定すると、そのブロックのテキストを読むことができます。

これで、定義、改良型コンパイラ、改良型ブロックハンドラ、改良型テキストエディタを提供する画面を書くことができ、アプリケーションを進めることができるようになりました。REMEMBER というエントリが欲しいのですが。今までは、手動で辞書をリセットすることができたので、必要なかったのです。

根でコードを変更することの難しさにお気づきでしょう。 強力なツールは、コアで辞書を移動させることができるようにすることです。ルートが絶対アドレスを使用しない場合、SHIFT エントリを定義してそれを使用します。 そうでなければ、絶対アドレスの数を最小限に抑え、それらを調整する、より精巧なSHIFT動詞を定義してください。

プログラムの SAVE には注意してください。念のため、新しいものを SAVE する前に、古いバージョンのバックアップをとっておいてください。

**Figure 1, Figure 2, Figure 3**

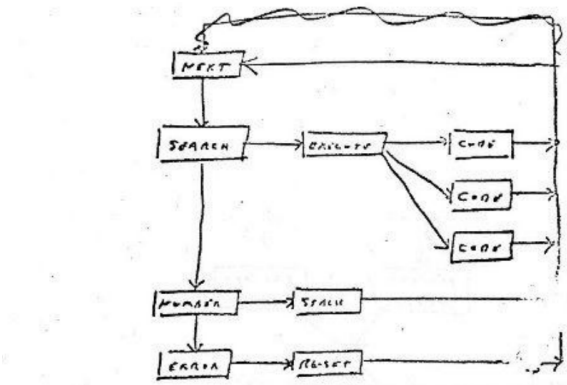



FIG. 1 CONTROL PROGRAM

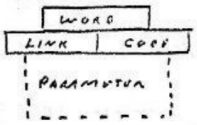



FIG 2 DICTIONARY ENTRY

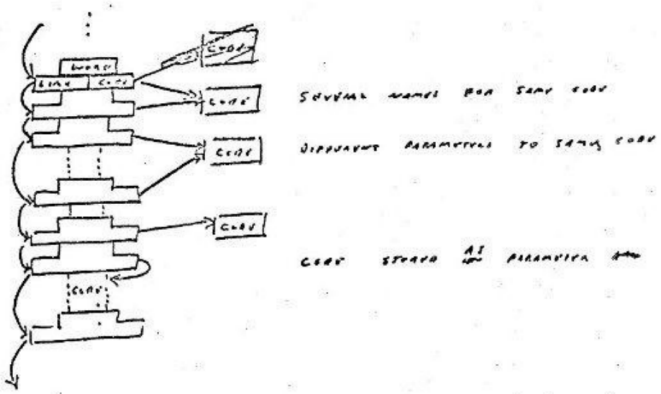



FIG 3 PORTION OF DICTIONARY

**Figure 6.2**

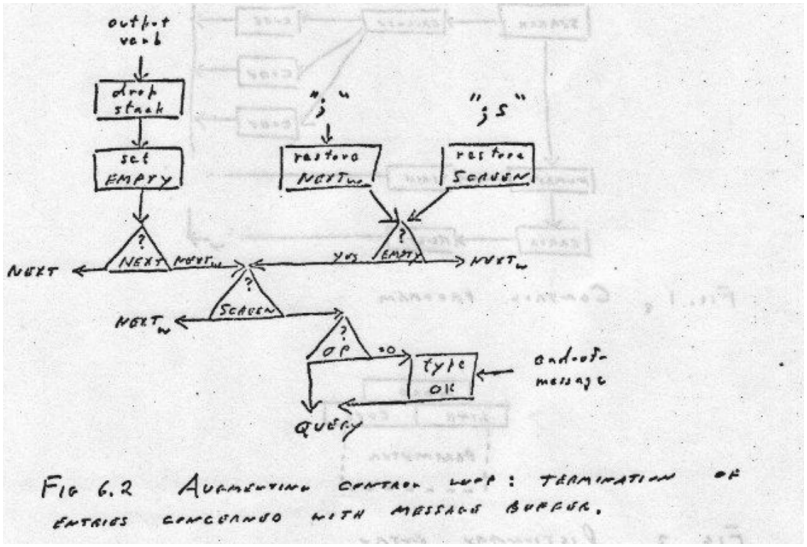
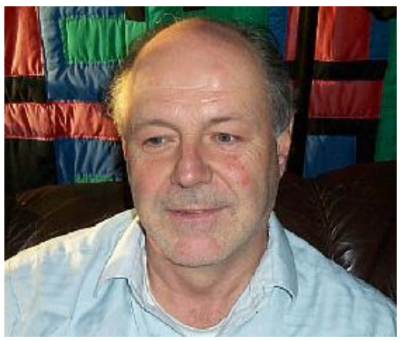



FIG 6.2 AUGMENTING CONTROL LOOP : TERMINATION OF ENTRIES CONCURRED WITH MESSAGE BUFFER.

# Charles H Moore



学歴: 1938年、ペンシルベニア州ピッツバーグ近郊のマッキースポート(McKeesport)で生まれる。ミシガン州フリント(Flint)で育ち、セントラル高校の卒業生総代(Validictorian)となる(1956年)。ナショナル・メリット・スカラシップでMITに入学し、カッパ・シグマ友愛会(Kappa Sigma fraternity)に入会。エクスプローラXIガンマ線衛星のデータ削減に関する論文で、1960年に物理学の理学学士号を取得。その後スタンフォード大学で2年間数学を学ぶ(1961年)。

プログラマ: ジョン・マッカーシーからLispを学ぶ。スミソニアン天体物理観測所でムーンウォッチ衛星の観測結果を予測するために、IBM704でFortran IIを使用(1958年)。このプログラムをアセンブラに圧縮し、衛星軌道を決定(1959年)。一方、スタンフォード線形加速器センターで電子ビームのステアリングを最適化するためにバロウズB5500用のAlgolを習得(1962年)。Charles H Moore and Associatesとして、タイムシェアリングサービスをサポートするためにFortran-Algolトランスレータを作成しました(1964年)。また、最初のミニコンピュータでリアルタイムのガスクロマトグラフをプログラミングした(1965年)。モハスコ社でオーダーエントリネットワークをプログラムするためにCobolを習得した(1968)。

Forth: Chuck は Forth を発明し (1968年)、彼の個人的なソフトウェアライブラリを IBM 1130 に集め、それを彼が見た最初のグラフィックターミナル (IBM 2250) に接続した。すぐに彼は、国立電波天文台のためにキットピークの30フィート望遠鏡を制御するためにForthを使いました (1970年)。

そして、エンジェル投資家からの5,000ドルでForth社の設立を支援(1973年)。その後10年間、数多くのミニ、マイクロ、メインフレームコンピュータにForthを移植した。そして、データベースからロボット工学まで、数多くのアプリケーションをプログラミングした。 1980年、Byte誌はThe Forth Languageの特集号を発行した。 Gregg Williamsの論説(2.5MB)は、Forthを外から見た珍しい見方を提供しています。

半導体(chips): ついに、Forth固有のアーキテクチャを実現するために、Forthチップを作ることを決意した。Novix, Incを創設し、C4000(1983年)をゲートアレイとして実装した。彼は、このチップを普及させるためのキットを開発・販売した。派生製品は最終的にHarris Semiconductor社に売却され、同社はこれを宇宙用RTX2000として販売しました(1988年)。

コンピュータ・カウボーイとして、彼はスタンダードセルのSh-Boomチップ(1985年)を設計し、その派生製品は現在も販売されています。そして、複数の専用プロセッサを搭載した「MuP21」(1990年)の設計ツールを独自に開発した。そして、ネットワークインターフェースを搭載した「F21」(1993年)。また、iTv社を創設し、インターネットアプリケーション向けに性能を向上させた同様のアーキテクチャであるi21(1996年)を設計しました。

コンピュータ・カウボーイに戻ったチャックは、colorForthを発明し、彼のVLSI設計ツールを移植し、チップ上で何度も複製できるシンプルなコアであるc18マイクロコンピュータ(2001年)を設計しました。彼のチップはどれも高性能と低消費電力を強調している。

最新の企業: http://www.greenarraychips.com/ 144個のコア・コントローラを搭載している。