LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

27 August 1989

Part XIII: PROCEDURES

INTRODUCTION

At last we get to the good part!

At this point we've studied almost all the basic features of compilers and parsing. We have learned how to translate arithmetic expressions, Boolean expressions, control constructs, data declarations, and I/O statements. We have defined a language, TINY 1.3, that embodies all these features, and we have written a rudimentary compiler that can translate them. By adding some file I/O we could indeed have a working compiler that could produce executable object files from programs written in TINY. With such a compiler, we could write simple programs that could read integer data, perform calculations with it, and output the results.

That's nice, but what we have is still only a toy language. We

can't read or write even a single character of text, and we still don't have procedures.

It's the features to be discussed in the next couple of installments that separate the men from the toys, so to speak. "Real" languages have more than one data type, and they support procedure calls. More than any others, it's these two features that give a language much of its character and personality. Once we have provided for them, our languages, TINY and its successors, will cease to become toys and will take on the character of real languages, suitable for serious programming jobs.

For several installments now, I've been promising you sessions on these two important subjects. Each time, other issues came up that required me to digress and deal with them. Finally, we've been able to put all those issues to rest and can get on with the mainstream of things. In this installment, I'll cover procedures. Next time, we'll talk about the basic data types.

ONE LAST DIGRESSION

This has been an extraordinarily difficult installment for me to write. The reason has nothing to do with the subject itself ... I've known what I wanted to say for some time, and in fact I presented most of this at Software Development '89, back in February. It has more to do with the approach. Let me explain.

When I first began this series, I told you that we would use several "tricks" to make things easy, and to let us learn the concepts without getting too bogged down in the details. Among these tricks was the idea of looking at individual pieces of a compiler at a time, i.e. performing experiments using the Cradle as a base. When we studied expressions, for example, we dealt with only that part of compiler theory. When we studied control structures, we wrote a different program, still based on the Cradle, to do that part. We only incorporated these concepts into a complete language fairly recently. These techniques have served us very well indeed, and led us to the development of a compiler for TINY version 1.3.

When I first began this session, I tried to build upon what we had already done, and just add the new features to the existing compiler. That turned out to be a little awkward and tricky ... much too much to suit me.

I finally figured out why. In this series of experiments, I had abandoned the very useful techniques that had allowed us to get here, and without meaning to I had switched over into a new method of working, that involved incremental changes to the full TINY compiler.

You need to understand that what we are doing here is a little unique. There have been a number of articles, such as the Small C articles by Cain and Hendrix, that presented finished compilers for one language or another. This is different. In this series of tutorials, you are watching me design and implement both a language and a compiler, in real time.

In the experiments that I've been doing in preparation for this article, I was trying to inject the changes into the TINY compiler in such a way that, at every step, we still had a real, working compiler. In other words, I was attempting an incremental enhancement of the language and its compiler, while at the same time explaining to you what I was doing.

That's a tough act to pull off! I finally realized that it was dumb to try. Having gotten this far using the idea of small experiments based on single-character tokens and simple, special-purpose programs, I had abandoned them in favor of working with the full compiler. It wasn't working.

So we're going to go back to our roots, so to speak. In this installment and the next, I'll be using single-character tokens again as we study the concepts of procedures, unfettered by the other baggage that we have accumulated in the previous sessions. As a matter of fact, I won't even attempt, at the end of this session, to merge the constructs into the TINY compiler. We'll save that for later.

After all this time, you don't need more buildup than that, so let's waste no more time and dive right in.

THE BASICS

All modern CPU's provide direct support for procedure calls, and the 68000 is no exception. For the 68000, the call is a BSR (PC-relative version) or JSR, and the return is RTS. All we have to do is to arrange for the compiler to issue these commands at the proper place.

Actually, there are really THREE things we have to address. One of them is the call/return mechanism. The second is the mechanism for DEFINING the procedure in the first place. And, finally, there is the issue of passing parameters to the called procedure. None of these things are really very difficult, and we can of course borrow heavily on what people have done in other languages ... there's no need to reinvent the wheel here. Of the three issues, that of parameter passing will occupy most of our attention, simply because there are so many options available.

A BASIS FOR EXPERIMENTS

As always, we will need some software to serve as a basis for what we are doing. We don't need the full TINY compiler, but we do need enough of a program so that some of the other constructs are present. Specifically, we need at least to be able to handle statements of some sort, and data declarations.

The program shown below is that basis. It's a vestigial form of TINY, with single-character tokens. It has data declarations, but only in their simplest form ... no lists or initializers. It has assignment statements, but only of the kind

```
<ident> = <ident>
```

In other words, the only legal expression is a single variable name. There are no control constructs ... the only legal statement is the assignment.

Most of the program is just the standard Cradle routines. I've shown the whole thing here, just to make sure we're all starting from the same point:

```
{--------------------------------------------------------------}
program Calls;

{--------------------------------------------------------------}
{ Constant Declarations }
```

```
const TAB = ^I;
      CR  = ^M;
      LF  = ^J;

{---------------------------------------------------------------}
{ Variable Declarations }

var Look: char;             { Lookahead Character }

var ST: Array['A'..'Z'] of char;


{---------------------------------------------------------------}
{ Read New Character From Input Stream }

procedure GetChar;
begin
   Read(Look);
end;

{---------------------------------------------------------------}
{ Report an Error }

procedure Error(s: string);
begin
   WriteLn;
   WriteLn(^G, 'Error: ', s, '.');
end;


{---------------------------------------------------------------}
{ Report Error and Halt }

procedure Abort(s: string);
begin
   Error(s);
   Halt;
end;


{---------------------------------------------------------------}
{ Report What Was Expected }

procedure Expected(s: string);
begin
   Abort(s + ' Expected');
end;


{---------------------------------------------------------------}
{ Report an Undefined Identifier }

procedure Undefined(n: string);
begin
   Abort('Undefined Identifier ' + n);
end;


{---------------------------------------------------------------}
{ Report an Duplicate Identifier }

procedure Duplicate(n: string);
begin
     Abort('Duplicate Identifier ' + n);
end;
```

```
{-------------------------------------------------------------}
{ Get Type of Symbol }

function TypeOf(n: char): char;
begin
     TypeOf := ST[n];
end;


{-------------------------------------------------------------}
{ Look for Symbol in Table }

function InTable(n: char): Boolean;
begin
   InTable := ST[n] <> ' ';
end;


{-------------------------------------------------------------}
{ Add a New Symbol to Table }

procedure AddEntry(Name, T: char);
begin
     if Intable(Name) then Duplicate(Name);
     ST[Name] := T;
end;


{-------------------------------------------------------------}
{ Check an Entry to Make Sure It's a Variable }

procedure CheckVar(Name: char);
begin
     if not InTable(Name) then Undefined(Name);
     if  TypeOf(Name)  <>  'v'  then    Abort(Name  +  ' is not a
variable');
end;


{-------------------------------------------------------------}
{ Recognize an Alpha Character }

function IsAlpha(c: char): boolean;
begin
   IsAlpha := upcase(c) in ['A'..'Z'];
end;


{-------------------------------------------------------------}
{ Recognize a Decimal Digit }

function IsDigit(c: char): boolean;
begin
   IsDigit := c in ['0'..'9'];
end;


{-------------------------------------------------------------}
{ Recognize an AlphaNumeric Character }

function IsAlNum(c: char): boolean;
begin
   IsAlNum := IsAlpha(c) or IsDigit(c);
end;


{-------------------------------------------------------------}
```

```
{ Recognize an Addop }

function IsAddop(c: char): boolean;
begin
   IsAddop := c in ['+', '-'];
end;


{---------------------------------------------------------------}
{ Recognize a Mulop }

function IsMulop(c: char): boolean;
begin
   IsMulop := c in ['*', '/'];
end;


{---------------------------------------------------------------}
{ Recognize a Boolean Orop }

function IsOrop(c: char): boolean;
begin
   IsOrop := c in ['|', '~'];
end;


{---------------------------------------------------------------}
{ Recognize a Relop }

function IsRelop(c: char): boolean;
begin
   IsRelop := c in ['=', '#', '<', '>'];
end;


{---------------------------------------------------------------}
{ Recognize White Space }

function IsWhite(c: char): boolean;
begin
   IsWhite := c in [' ', TAB];
end;


{---------------------------------------------------------------}
{ Skip Over Leading White Space }

procedure SkipWhite;
begin
   while IsWhite(Look) do
      GetChar;
end;


{---------------------------------------------------------------}
{ Skip Over an End-of-Line }

procedure Fin;
begin
   if Look = CR then begin
      GetChar;
      if Look = LF then
         GetChar;
   end;
end;
```

```
{---------------------------------------------------------------}
{ Match a Specific Input Character }

procedure Match(x: char);
begin
   if Look = x then GetChar
     else Expected('''' + x + '''');
     SkipWhite;
end;


{---------------------------------------------------------------}
{ Get an Identifier }

function GetName: char;
begin
   if not IsAlpha(Look) then Expected('Name');
   GetName := UpCase(Look);
     GetChar;
     SkipWhite;
end;


{---------------------------------------------------------------}
{ Get a Number }

function GetNum: char;
begin
   if not IsDigit(Look) then Expected('Integer');
   GetNum := Look;
     GetChar;
     SkipWhite;
end;


{---------------------------------------------------------------}
{ Output a String with Tab }

procedure Emit(s: string);
begin
   Write(TAB, s);
end;


{---------------------------------------------------------------}
{ Output a String with Tab and CRLF }

procedure EmitLn(s: string);
begin
   Emit(s);
   WriteLn;
end;


{---------------------------------------------------------------}
{ Post a Label To Output }

procedure PostLabel(L: string);
begin
   WriteLn(L, ':');
end;


{---------------------------------------------------------------}
{ Load a Variable to the Primary Register }

procedure LoadVar(Name: char);
```

```
begin
     CheckVar(Name);
     EmitLn('MOVE ' + Name + '(PC),D0');
end;


{---------------------------------------------------------------}
{ Store the Primary Register }

procedure StoreVar(Name: char);
begin
     CheckVar(Name);
     EmitLn('LEA ' + Name + '(PC),A0');
   EmitLn('MOVE D0,(A0)')
end;


{---------------------------------------------------------------}
{ Initialize }

procedure Init;
var i: char;
begin
     GetChar;
     SkipWhite;
     for i := 'A' to 'Z' do
          ST[i] := ' ';
end;


{---------------------------------------------------------------}
{ Parse and Translate an Expression }
{ Vestigial Version }

procedure Expression;
begin
     LoadVar(GetName);
end;


{---------------------------------------------------------------}
{ Parse and Translate an Assignment Statement }

procedure Assignment;
var Name: char;
begin
     Name := GetName;
     Match('=');
     Expression;
     StoreVar(Name);
end;


{---------------------------------------------------------------}









{ Parse and Translate a Block of Statements }

procedure DoBlock;
begin
     while not(Look in ['e']) do begin
```

```
          Assignment;
          Fin;
   end;
end;


{---------------------------------------------------------------}
{ Parse and Translate a Begin-Block }

procedure BeginBlock;
begin
     Match('b');
     Fin;
     DoBlock;
     Match('e');
     Fin;
end;


{---------------------------------------------------------------}
{ Allocate Storage for a Variable }

procedure Alloc(N: char);
begin
     if InTable(N) then Duplicate(N);
   ST[N] := 'v';
     WriteLn(N, ':', TAB, 'DC 0');
end;


{---------------------------------------------------------------}
{ Parse and Translate a Data Declaration }

procedure Decl;
var Name: char;
begin
   Match('v');
     Alloc(GetName);
end;


{---------------------------------------------------------------}
{ Parse and Translate Global Declarations }

procedure TopDecls;
begin
     while Look <> 'b' do begin
      case Look of
        'v': Decl;
      else Abort('Unrecognized Keyword ' + Look);
          end;
          Fin;
     end;
end;


{---------------------------------------------------------------}
{ Main Program }

begin
     Init;
     TopDecls;
     BeginBlock;
end.
{---------------------------------------------------------------}
```

Note that we DO have a symbol table, and there is logic to check a variable name to make sure it's a legal one. It's also worth noting that I have included the code you've seen before to provide for white space and newlines. Finally, note that the main program is delimited, as usual, by BEGIN-END brackets.

Once you've copied the program to Turbo, the first step is to compile it and make sure it works. Give it a few declarations, and then a begin-block. Try something like:

```
va              (for VAR A)
vb              (for VAR B)
vc              (for VAR C)
b               (for BEGIN)
a=b
b=c
e.              (for END.)
```

As usual, you should also make some deliberate errors, and verify that the program catches them correctly.

DECLARING A PROCEDURE

If you're satisfied that our little program works, then it's time to deal with the procedures. Since we haven't talked about parameters yet, we'll begin by considering only procedures that have no parameter lists.

As a start, let's consider a simple program with a procedure, and think about the code we'd like to see generated for it:

```
PROGRAM FOO;
.
.
PROCEDURE BAR;                      BAR:
BEGIN                                    .
.                                        .
.                                        .
END;                                     RTS

BEGIN { MAIN PROGRAM }              MAIN:
.                                        .
.                                        .
FOO;                                     BSR BAR
.                                        .
.                                        .
END.                                     END MAIN
```

Here I've shown the high-order language constructs on the left, and the desired assembler code on the right. The first thing to notice is that we certainly don't have much code to generate here! For the great bulk of both the procedure and the main program, our existing constructs take care of the code to be generated.

The key to dealing with the body of the procedure is to recognize that although a procedure may be quite long, declaring it is really no different than declaring a variable. It's just one more kind of declaration. We can write the BNF:

```
<declaration> ::= <data decl> | <procedure>
```

This means that it should be easy to modify TopDecl to deal with procedures. What about the syntax of a procedure? Well, here's a suggested syntax, which is essentially that of Pascal:

```
<procedure> ::= PROCEDURE <ident> <begin-block>
```

There is practically no code generation required, other than that generated within the begin-block. We need only emit a label at the beginning of the procedure, and an RTS at the end.

Here's the required code:

```
{-------------------------------------------------------------}
{ Parse and Translate a Procedure Declaration }

procedure DoProc;
var N: char;
begin
     Match('p');
     N := GetName;
     Fin;
     if InTable(N) then Duplicate(N);
     ST[N] := 'p';
     PostLabel(N);
     BeginBlock;
     Return;
end;
{-------------------------------------------------------------}
```

Note that I've added a new code generation routine, Return, which merely emits an RTS instruction. The creation of that routine is "left as an exercise for the student."

To finish this version, add the following line within the Case statement in DoBlock:

```
            'p': DoProc;
```

I should mention that this structure for declarations, and the BNF that drives it, differs from standard Pascal. In the Jensen & Wirth definition of Pascal, variable declarations, in fact ALL kinds of declarations, must appear in a specific sequence, i.e. labels, constants, types, variables, procedures, and main program. To follow such a scheme, we should separate the two declarations, and have code in the main program something like

```
     DoVars;
     DoProcs;
     DoMain;
```

However, most implementations of Pascal, including Turbo, don't

require that order and let you freely mix up the various declarations, as long as you still don't try to refer to something before it's declared. Although it may be more aesthetically pleasing to declare all the global variables at the top of the program, it certainly doesn't do any HARM to allow them to be sprinkled around. In fact, it may do some GOOD, in the sense that it gives you the opportunity to do a little rudimentary information hiding. Variables that should be accessed only by the main program, for example, can be declared just before it and will thus be inaccessible by the procedures.

OK, try this new version out. Note that we can declare as many procedures as we choose (as long as we don't run out of single-character names!), and the labels and RTS's all come out in the right places.

It's worth noting here that I do _NOT_ allow for nested procedures. In TINY, all procedures must be declared at the global level, the same as in C. There has been quite a discussion about this point in the Computer Language Forum of CompuServe. It turns out that there is a significant penalty in complexity that must be paid for the luxury of nested procedures. What's more, this penalty gets paid at RUN TIME, because extra code must be added and executed every time a procedure is called. I also happen to believe that nesting is not a good idea, simply on the grounds that I have seen too many abuses of the feature. Before going on to the next step, it's also worth noting that the "main program" as it stands is incomplete, since it doesn't have the label and END statement. Let's fix that little oversight:

```
{--------------------------------------------------------------}
{ Parse and Translate a Main Program }

procedure DoMain;
begin
     Match('b');
     Fin;
     Prolog;
     DoBlock;
     Epilog;
end;
{--------------------------------------------------------------}
.
.
.
{--------------------------------------------------------------}
{ Main Program }

begin
     Init;
     TopDecls;
     DoMain;
end.
{--------------------------------------------------------------}
```

Note that DoProc and DoMain are not quite symmetrical. DoProc uses a call to BeginBlock, whereas DoMain cannot. That's because a procedure is signaled by the keyword PROCEDURE (abbreviated by a 'p' here), while the main program gets no keyword other than the BEGIN itself.

And _THAT_ brings up an interesting question: WHY?

If we look at the structure of C programs, we find that all functions are treated just alike, except that the main program

happens to be identified by its name, "main." Since C functions can appear in any order, the main program can also be anywhere in the compilation unit.

In Pascal, on the other hand, all variables and procedures must be declared before they're used, which means that there is no point putting anything after the main program ... it could never be accessed. The "main program" is not identified at all, other than being that part of the code that comes after the global BEGIN. In other words, if it ain't anything else, it must be the main program.

This causes no small amount of confusion for beginning programmers, and for big Pascal programs sometimes it's difficult to find the beginning of the main program at all. This leads to conventions such as identifying it in comments:

```
BEGIN { of MAIN }
```

This has always seemed to me to be a bit of a kludge. The question comes up: Why should the main program be treated so much differently than a procedure? In fact, now that we've recognized that procedure declarations are just that ... part of the global declarations ... isn't the main program just one more declaration, also?

The answer is yes, and by treating it that way, we can simplify the code and make it considerably more orthogonal. I propose that we use an explicit keyword, PROGRAM, to identify the main program (Note that this means that we can't start the file with it, as in Pascal). In this case, our BNF becomes:

```
<declaration> ::= <data decl> | <procedure> | <main program>

<procedure> ::= PROCEDURE <ident> <begin-block>

<main program> ::= PROGRAM <ident> <begin-block>
```

The code also looks much better, at least in the sense that DoMain and DoProc look more alike:

```
{------------------------------------------------------------}
{ Parse and Translate a Main Program }

procedure DoMain;
var N: char;
begin
     Match('P');
     N := GetName;
     Fin;
     if InTable(N) then Duplicate(N);
     Prolog;
     BeginBlock;
end;
{------------------------------------------------------------}
.
.
.
{------------------------------------------------------------}
{ Parse and Translate Global Declarations }
```

```
procedure TopDecls;
begin
     while Look <> '.' do begin
      case Look of
            'v': Decl;
            'p': DoProc;
            'P': DoMain;
          else Abort('Unrecognized Keyword ' + Look);
          end;
          Fin;
     end;
end;


{---------------------------------------------------------------}
{ Main Program }

begin
     Init;
     TopDecls;
     Epilog;
end.
{---------------------------------------------------------------}
```

Since the declaration of the main program is now within the loop of TopDecl, that does present some difficulties. How do we ensure that it's the last thing in the file? And how do we ever exit from the loop? My answer for the second question, as you can see, was to bring back our old friend the period. Once the parser sees that, we're done.

To answer the first question: it depends on how far we're willing to go to protect the programmer from dumb mistakes. In the code that I've shown, there's nothing to keep the programmer from adding code after the main program ... even another main program. The code will just not be accessible. However, we COULD access it via a FORWARD statement, which we'll be providing later. As a matter of fact, many assembler language programmers like to use the area just after the program to declare large, uninitialized data blocks, so there may indeed be some value in not requiring the main program to be last. We'll leave it as it is.

If we decide that we should give the programmer a little more help than that, it's pretty easy to add some logic to kick us out of the loop once the main program has been processed. Or we could at least flag an error if someone tries to include two mains.

CALLING THE PROCEDURE

If you're satisfied that things are working, let's address the second half of the equation ... the call.

Consider the BNF for a procedure call:

```
     <proc_call> ::= <identifier>
```

for an assignment statement, on the other hand, the BNF is:

```
     <assignment> ::= <identifier> '=' <expression>
```

At this point we seem to have a problem. The two BNF statements both begin on the right-hand side with the token <identifier>. How are we supposed to know, when we see the identifier, whether we have a procedure call or an assignment statement? This looks like a case where our parser ceases being predictive, and indeed that's exactly the case. However, it turns out to be an easy problem to fix, since all we have to do is to look at the type of the identifier, as recorded in the symbol table. As we've discovered before, a minor local violation of the predictive parsing rule can be easily handled as a special case.

Here's how to do it:

```
{--------------------------------------------------------------}
{ Parse and Translate an Assignment Statement }

procedure Assignment(Name: char);
begin
     Match('=');
     Expression;
     StoreVar(Name);
end;


{--------------------------------------------------------------}
{ Decide if a Statement is an Assignment or Procedure Call }

procedure AssignOrProc;
var Name: char;
begin
     Name := GetName;
     case TypeOf(Name) of
          ' ': Undefined(Name);
          'v': Assignment(Name);
          'p': CallProc(Name);
          else Abort('Identifier ' + Name +
                                ' Cannot Be Used Here');
     end;
end;


{--------------------------------------------------------------}
{ Parse and Translate a Block of Statements }

procedure DoBlock;
begin
     while not(Look in ['e']) do begin
          AssignOrProc;
          Fin;
   end;
end;
{--------------------------------------------------------------}
```

As you can see, procedure Block now calls AssignOrProc instead of Assignment. The function of this new procedure is to simply read the identifier, determine its type, and then call whichever procedure is appropriate for that type. Since the name has already been read, we must pass it to the two procedures, and modify Assignment to match. Procedure CallProc is a simple code generation routine:

```
{--------------------------------------------------------------}
```

```
{ Call a Procedure }

procedure CallProc(N: char);
begin
     EmitLn('BSR ' + N);
end;
{--------------------------------------------------------------}
```

Well, at this point we have a compiler that can deal with procedures. It's worth noting that procedures can call procedures to any depth. So even though we don't allow nested DECLARATIONS, there is certainly nothing to keep us from nesting CALLS, just as we would expect to do in any language. We're getting there, and it wasn't too hard, was it?

Of course, so far we can only deal with procedures that have no parameters. The procedures can only operate on the global variables by their global names. So at this point we have the equivalent of BASIC's GOSUB construct. Not too bad ... after all lots of serious programs were written using GOSUBs, but we can do better, and we will. That's the next step.

PASSING PARAMETERS

Again, we all know the basic idea of passed parameters, but let's review them just to be safe.

In general the procedure is given a parameter list, for example

```
     PROCEDURE FOO(X, Y, Z)
```

In the declaration of a procedure, the parameters are called formal parameters, and may be referred to in the body of the procedure by those names. The names used for the formal parameters are really arbitrary. Only the position really counts. In the example above, the name 'X' simply means "the first parameter" wherever it is used.

When a procedure is called, the "actual parameters" passed to it are associated with the formal parameters, on a one-for-one basis.

The BNF for the syntax looks something like this:

```
     <procedure> ::= PROCEDURE <ident>
                     '(' <param-list> ')' <begin-block>


     <param_list> ::= <parameter> ( ',' <parameter> )* | null
```

Similarly, the procedure call looks like:

```
     <proc call> ::= <ident> '(' <param-list> ')'
```

Note that there is already an implicit decision built into this syntax. Some languages, such as Pascal and Ada, permit parameter lists to be optional. If there are no parameters, you simply leave off the parens completely. Other languages, like C and Modula 2, require the parens even if the list is empty. Clearly, the example we just finished corresponds to the former point of view. But to tell the truth I prefer the latter. For procedures alone, the decision would seem to favor the "listless" approach.

The statement

```
     Initialize; ,
```

standing alone, can only mean a procedure call. In the parsers we've been writing, we've made heavy use of parameterless procedures, and it would seem a shame to have to write an empty pair of parens for each case.

But later on we're going to be using functions, too. And since functions can appear in the same places as simple scalar identifiers, you can't tell the difference between the two. You have to go back to the declarations to find out. Some folks consider this to be an advantage. Their argument is that an identifier gets replaced by a value, and what do you care whether it's done by substitution or by a function? But we sometimes _DO_ care, because the function may be quite time-consuming. If, by writing a simple identifier into a given expression, we can incur a heavy run-time penalty, it seems to me we ought to be made aware of it.

Anyway, Niklaus Wirth designed both Pascal and Modula 2. I'll give him the benefit of the doubt and assume that he had a good reason for changing the rules the second time around!

Needless to say, it's an easy thing to accomodate either point of view as we design a language, so this one is strictly a matter of personal preference. Do it whichever way you like best.

Before we go any further, let's alter the translator to handle a (possibly empty) parameter list. For now we won't generate any extra code ... just parse the syntax. The code for processing the declaration has very much the same form we've seen before when dealing with VAR-lists:

```
{--------------------------------------------------------------}
{ Process the Formal Parameter List of a Procedure }

procedure FormalList;
begin
     Match('(');
     if Look <> ')' then begin
          FormalParam;
          while Look = ',' do begin
               Match(',');
               FormalParam;
          end;
     end;
     Match(')');
end;
{--------------------------------------------------------------}
```

Procedure DoProc needs to have a line added to call FormalList:

```
{--------------------------------------------------------------}
{ Parse and Translate a Procedure Declaration }

procedure DoProc;
var N: char;
begin
     Match('p');
     N := GetName;
```

```
     FormalList;
     Fin;
     if InTable(N) then Duplicate(N);
     ST[N] := 'p';
     PostLabel(N);
     BeginBlock;
     Return;
end;
{--------------------------------------------------------------}
```

For now, the code for FormalParam is just a dummy one that simply skips the parameter name:

```
{--------------------------------------------------------------}
{ Process a Formal Parameter }

procedure FormalParam;
var Name:  char;
begin
     Name := GetName;
end;
{--------------------------------------------------------------}
```

For the actual procedure call, there must be similar code to process the actual parameter list:

```
{--------------------------------------------------------------}
{ Process an Actual Parameter }

procedure Param;
var Name:  char;
begin
     Name := GetName;
end;


{--------------------------------------------------------------}
{ Process the Parameter List for a Procedure  Call }

procedure ParamList;
begin
     Match('(');
     if Look <> ')' then begin
          Param;
          while Look = ',' do begin
               Match(',');
               Param;
          end;
     end;
     Match(')');
end;


{--------------------------------------------------------------}
{ Process a Procedure Call }

procedure CallProc(Name: char);
begin
     ParamList;
     Call(Name);
end;
{--------------------------------------------------------------}
```

Note here that CallProc is no longer just a simple code generation routine. It has some structure to it. To handle this, I've renamed the code generation routine to just Call, and called it from within CallProc.

OK, if you'll add all this code to your translator and try it out, you'll find that you can indeed parse the syntax properly. I'll note in passing that there is _NO_ checking to make sure that the number (and, later, types) of formal and actual parameters match up. In a production compiler, we must of course do this. We'll ignore the issue now if for no other reason than that the structure of our symbol table doesn't currently give us a place to store the necessary information. Later on, we'll have a place for that data and we can deal with the issue then.

THE SEMANTICS OF PARAMETERS

So far we've dealt with the SYNTAX of parameter passing, and we've got the parsing mechanisms in place to handle it. Next, we have to look at the SEMANTICS, i.e., the actions to be taken when we encounter parameters. This brings us square up against the issue of the different ways parameters can be passed.

There is more than one way to pass a parameter, and the way we do it can have a profound effect on the character of the language. So this is another of those areas where I can't just give you my solution. Rather, it's important that we spend some time looking at the alternatives so that you can go another route if you choose to.

There are two main ways parameters are passed:

- By value
- By reference (address)

The differences are best seen in the light of a little history.

The old FORTRAN compilers passed all parameters by reference. In other words, what was actually passed was the address of the parameter. This meant that the called subroutine was free to either read or write that parameter, as often as it chose to, just as though it were a global variable. This was actually quite an efficient way to do things, and it was pretty simple since the same mechanism was used in all cases, with one exception that I'll get to shortly.

There were problems, though. Many people felt that this method created entirely too much coupling between the called subroutine and its caller. In effect, it gave the subroutine complete access to all variables that appeared in the parameter list.

Many times, we didn't want to actually change a parameter, but only use it as an input. For example, we might pass an element count to a subroutine, and wish we could then use that count within a DO-loop. To avoid changing the value in the calling program, we had to make a local copy of the input parameter, and operate only on the copy. Some FORTRAN programmers, in fact, made it a practice to copy ALL parameters except those that were to be used as return values. Needless to say, all this copying defeated a good bit of the efficiency associated with the approach.

There was, however, an even more insidious problem, which was not really just the fault of the "pass by reference" convention, but a bad convergence of several implementation decisions.

Suppose we have a subroutine:

```
SUBROUTINE FOO(X, Y, N)
```

where N is some kind of input count or flag. Many times, we'd like to be able to pass a literal or even an expression in place of a variable, such as:

```
CALL FOO(A, B, J + 1)
```

Here the third parameter is not a variable, and so it has no address. The earliest FORTRAN compilers did not allow such things, so we had to resort to subterfuges like:

```
K = J + 1
CALL FOO(A, B, K)
```

Here again, there was copying required, and the burden was on the programmer to do it. Not good.

Later FORTRAN implementations got rid of this by allowing expressions as parameters. What they did was to assign a compiler-generated variable, store the value of the expression in the variable, and then pass the address of the expression.

So far, so good. Even if the subroutine mistakenly altered the anonymous variable, who was to know or care? On the next call, it would be recalculated anyway.

The problem arose when someone decided to make things more efficient. They reasoned, rightly enough, that the most common kind of "expression" was a single integer value, as in:

```
CALL FOO(A, B, 4)
```

It seemed inefficient to go to the trouble of "computing" such an integer and storing it in a temporary variable, just to pass it through the calling list. Since we had to pass the address of the thing anyway, it seemed to make lots of sense to just pass the address of the literal integer, 4 in the example above.

To make matters more interesting, most compilers, then and now, identify all literals and store them separately in a "literal pool," so that we only have to store one value for each unique literal. That combination of design decisions: passing expressions, optimization for literals as a special case, and use of a literal pool, is what led to disaster.

To see how it works, imagine that we call subroutine FOO as in the example above, passing it a literal 4. Actually, what gets passed is the address of the literal 4, which is stored in the literal pool. This address corresponds to the formal parameter, K, in the subroutine itself.

Now suppose that, unbeknownst to the programmer, subroutine FOO actually modifies K to be, say, -7. Suddenly, that literal 4 in the literal pool gets CHANGED, to a -7. From then on, every expression that uses a 4 and every subroutine that passes a 4

will be using the value of -7 instead! Needless to say, this can lead to some bizarre and difficult-to-find behavior. The whole thing gave the concept of pass-by-reference a bad name, although as we have seen, it was really a combination of design decisions that led to the problem.

In spite of the problem, the FORTRAN approach had its good points. Chief among them is the fact that we don't have to support multiple mechanisms. The same scheme, passing the address of the argument, works for EVERY case, including arrays. So the size of the compiler can be reduced.

Partly because of the FORTRAN gotcha, and partly just because of the reduced coupling involved, modern languages like C, Pascal, Ada, and Modula 2 generally pass scalars by value.

This means that the value of the scalar is COPIED into a separate value used only for the call. Since the value passed is a copy, the called procedure can use it as a local variable and modify it any way it likes. The value in the caller will not be changed.

It may seem at first that this is a bit inefficient, because of the need to copy the parameter. But remember that we're going to have to fetch SOME value to pass anyway, whether it be the parameter itself or an address for it. Inside the subroutine, using pass-by-value is definitely more efficient, since we eliminate one level of indirection. Finally, we saw earlier that with FORTRAN, it was often necessary to make copies within the subroutine anyway, so pass-by-value reduces the number of local variables. All in all, pass-by-value is better.

Except for one small little detail: if all parameters are passed by value, there is no way for a called to procedure to return a result to its caller! The parameter passed is NOT altered in the caller, only in the called procedure. Clearly, that won't get the job done.

There have been two answers to this problem, which are equivalent. In Pascal, Wirth provides for VAR parameters, which are passed-by-reference. What a VAR parameter is, in fact, is none other than our old friend the FORTRAN parameter, with a new name and paint job for disguise. Wirth neatly gets around the "changing a literal" problem as well as the "address of an expression" problem, by the simple expedient of allowing only a variable to be the actual parameter. In other words, it's the same restriction that the earliest FORTRANs imposed.

C does the same thing, but explicitly. In C, _ALL_ parameters are passed by value. One kind of variable that C supports, however, is the pointer. So by passing a pointer by value, you in effect pass what it points to by reference. In some ways this works even better yet, because even though you can change the variable pointed to all you like, you still CAN'T change the pointer itself. In a function such as strcpy, for example, where the pointers are incremented as the string is copied, we are really only incrementing copies of the pointers, so the values of those pointers in the calling procedure still remain as they were. To modify a pointer, you must pass a pointer to the pointer.

Since we are simply performing experiments here, we'll look at BOTH pass-by-value and pass-by-reference. That way, we'll be able to use either one as we need to. It's worth mentioning that it's going to be tough to use the C approach to pointers here, since a pointer is a different type and we haven't studied types yet!

PASS-BY-VALUE

Let's just try some simple-minded things and see where they lead us. Let's begin with the pass-by-value case. Consider the procedure call:

```
FOO(X, Y)
```

Almost the only reasonable way to pass the data is through the CPU stack. So the code we'd like to see generated might look something like this:

```
MOVE X(PC),-(SP)    ; Push X
MOVE Y(PC),-(SP)    ; Push Y
BSR FOO             ; Call FOO
```

That certainly doesn't seem too complex!

When the BSR is executed, the CPU pushes the return address onto the stack and jumps to FOO. At this point the stack will look like this:

```
          .
          .
          Value of X (2 bytes)
          Value of Y (2 bytes)
  SP -->  Return Address (4 bytes)
```

So the values of the parameters have addresses that are fixed offsets from the stack pointer. In this example, the addresses are:

```
X:  6(SP)
Y:  4(SP)
```

Now consider what the called procedure might look like:

```
PROCEDURE FOO(A, B)
BEGIN
     A = B
END
```

(Remember, the names of the formal parameters are arbitrary ... only the positions count.)

The desired output code might look like:

```
FOO: MOVE 4(SP),D0
     MOVE D0,6(SP)
     RTS
```

Note that, in order to address the formal parameters, we're going to have to know which position they have in the parameter list. This means some changes to the symbol table stuff. In fact, for our single-character case it's best to just create a new symbol table for the formal parameters.

Let's begin by declaring a new table:

```
     var Params: Array['A'..'Z'] of integer;
```

We also will need to keep track of how many parameters a given procedure has:

```
     var NumParams: integer;
```

And we need to initialize the new table. Now, remember that the formal parameter list will be different for each procedure that we process, so we'll need to initialize that table anew for each procedure. Here's the initializer:

```
{-------------------------------------------------------------}
{ Initialize Parameter Table to Null }

procedure ClearParams;
var i: char;
begin
     for i := 'A' to 'Z' do
          Params[i] := 0;
     NumParams := 0;
end;
{-------------------------------------------------------------}
```

We'll put a call to this procedure in Init, and also at the end of DoProc:

```
{-------------------------------------------------------------}
{ Initialize }

procedure Init;
var i: char;
begin
     GetChar;
     SkipWhite;
     for i := 'A' to 'Z' do
          ST[i] := ' ';
     ClearParams;
end;
{-------------------------------------------------------------}
.
.
.
{-------------------------------------------------------------}
{ Parse and Translate a Procedure Declaration }

procedure DoProc;
var N: char;
begin
     Match('p');
     N := GetName;
     FormalList;
     Fin;
     if InTable(N) then Duplicate(N);
     ST[N] := 'p';
     PostLabel(N);
     BeginBlock;
```

```
     Return;
     ClearParams;
end;
{-------------------------------------------------------------}
```

Note that the call within DoProc ensures that the table will be clear when we're in the main program.

OK, now we need a few procedures to work with the table. The next few functions are essentially copies of InTable, TypeOf, etc.:

```
{-------------------------------------------------------------}
{ Find the Parameter Number }

function ParamNumber(N: char): integer;
begin
     ParamNumber := Params[N];
end;


{-------------------------------------------------------------}
{ See if an Identifier is a Parameter }

function IsParam(N: char): boolean;
begin
     IsParam := Params[N] <> 0;
end;


{-------------------------------------------------------------}
{ Add a New Parameter to Table }

procedure AddParam(Name: char);
begin
     if IsParam(Name) then Duplicate(Name);
     Inc(NumParams);
     Params[Name] := NumParams;
end;
{-------------------------------------------------------------}
```

Finally, we need some code generation routines:

```
{-------------------------------------------------------------}
{ Load a Parameter to the Primary Register }

procedure LoadParam(N: integer);
var Offset: integer;
begin
     Offset := 4 + 2 * (NumParams - N);
     Emit('MOVE ');
     WriteLn(Offset, '(SP),D0');
end;


{-------------------------------------------------------------}
{ Store a Parameter from the Primary Register }

procedure StoreParam(N: integer);
var Offset: integer;
begin
     Offset := 4 + 2 * (NumParams - N);
```

```
     Emit('MOVE D0,');
     WriteLn(Offset, '(SP)');
end;


{---------------------------------------------------------------}
{ Push The Primary Register to the Stack }

procedure Push;
begin
     EmitLn('MOVE D0,-(SP)');
end;
{---------------------------------------------------------------}
```

(The last routine is one we've seen before, but it wasn't in this vestigial version of the program.)

With those preliminaries in place, we're ready to deal with the semantics of procedures with calling lists (remember, the code to deal with the syntax is already in place).

Let's begin by processing a formal parameter. All we have to do is to add each parameter to the parameter symbol table:

```
{---------------------------------------------------------------}
{ Process a Formal Parameter }

procedure FormalParam;
begin
     AddParam(GetName);
end;
{---------------------------------------------------------------}
```

Now, what about dealing with a formal parameter when it appears in the body of the procedure? That takes a little more work. We must first determine that it IS a formal parameter. To do this, I've written a modified version of TypeOf:

```
{---------------------------------------------------------------}
{ Get Type of Symbol }

function TypeOf(n: char): char;
begin
     if IsParam(n) then
          TypeOf := 'f'
     else
          TypeOf := ST[n];
end;
{---------------------------------------------------------------}
```

(Note that, since TypeOf now calls IsParam, it may need to be relocated in your source.)

We also must modify AssignOrProc to deal with this new type:

```
{---------------------------------------------------------------}
{ Decide if a Statement is an Assignment or Procedure Call }

procedure AssignOrProc;
var Name: char;
begin
```

```
     Name := GetName;
     case TypeOf(Name) of
          ' ': Undefined(Name);
          'v', 'f': Assignment(Name);
          'p': CallProc(Name);
          else  Abort('Identifier ' + Name +  '  Cannot  Be  Used
Here');
     end;
end;
{--------------------------------------------------------------}
```

Finally, the code to process an assignment statement and an expression must be extended:

```
{--------------------------------------------------------------}
{ Parse and Translate an Expression }
{ Vestigial Version }

procedure Expression;
var Name: char;
begin
     Name := GetName;
     if IsParam(Name) then
          LoadParam(ParamNumber(Name))
     else
          LoadVar(Name);
end;


{--------------------------------------------------------------}
{ Parse and Translate an Assignment Statement }

procedure Assignment(Name: char);
begin
     Match('=');
     Expression;
     if IsParam(Name) then
          StoreParam(ParamNumber(Name))
     else
          StoreVar(Name);
end;
{--------------------------------------------------------------}
```

As you can see, these procedures will treat every variable name encountered as either a formal parameter or a global variable, depending on whether or not it appears in the parameter symbol table. Remember that we are using only a vestigial form of Expression. In the final program, the change shown here will have to be added to Factor, not Expression.

The rest is easy. We need only add the semantics to the actual procedure call, which we can do with one new line of code:

```
{--------------------------------------------------------------}
{ Process an Actual Parameter }

procedure Param;
begin
     Expression;
     Push;
end;
{--------------------------------------------------------------}
```

That's it. Add these changes to your program and give it a try. Try declaring one or two procedures, each with a formal parameter list. Then do some assignments, using combinations of global and formal parameters. You can call one procedure from within another, but you cannot DECLARE a nested procedure. You can even pass formal parameters from one procedure to another. If we had the full syntax of the language here, you'd also be able to do things like read or write formal parameters or use them in complicated expressions.

WHAT'S WRONG?

At this point, you might be thinking: Surely there's more to this than a few pushes and pops. There must be more to passing parameters than this.

You'd be right. As a matter of fact, the code that we're generating here leaves a lot to be desired in several respects.

The most glaring oversight is that it's wrong! If you'll look back at the code for a procedure call, you'll see that the caller pushes each actual parameter onto the stack before it calls the procedure. The procedure USES that information, but it doesn't change the stack pointer. That means that the stuff is still there when we return. SOMEBODY needs to clean up the stack, or we'll soon be in very hot water!

Fortunately, that's easily fixed. All we have to do is to increment the stack pointer when we're finished.

Should we do that in the calling program, or the called procedure? Some folks let the called procedure clean up the stack, since that requires less code to be generated per call, and since the procedure, after all, knows how many parameters it's got. But that means that it must do something with the return address so as not to lose it.

I prefer letting the caller clean up, so that the callee need only execute a return. Also, it seems a bit more balanced, since the caller is the one who "messed up" the stack in the first place. But THAT means that the caller must remember how many items it pushed. To make things easy, I've modified the procedure ParamList to be a function instead of a procedure, returning the number of bytes pushed:

```
{---------------------------------------------------------------}
{ Process the Parameter List for a Procedure  Call }

function ParamList: integer;
var N: integer;
begin
     N := 0;
     Match('(');
     if Look <> ')' then begin
          Param;
          inc(N);
          while Look = ',' do begin
               Match(',');
               Param;
               inc(N);
          end;
     end;
     Match(')');
     ParamList := 2 * N;
```

```
end;
{---------------------------------------------------------------}
```

Procedure CallProc then uses this to clean up the stack:

```
{---------------------------------------------------------------}
{ Process a Procedure Call }

procedure CallProc(Name: char);
var N: integer;
begin
     N := ParamList;
     Call(Name);
     CleanStack(N);
end;
{---------------------------------------------------------------}
```

Here I've created yet another code generation procedure:

```
{---------------------------------------------------------------}
{ Adjust the Stack Pointer Upwards by N Bytes }

procedure CleanStack(N: integer);
begin
     if N > 0 then begin
          Emit('ADD #');
          WriteLn(N, ',SP');
     end;
end;
{---------------------------------------------------------------}
```

OK, if you'll add this code to your compiler, I think you'll find that the stack is now under control.

The next problem has to do with our way of addressing relative to the stack pointer. That works fine in our simple examples, since with our rudimentary form of expressions nobody else is messing with the stack. But consider a different example as simple as:

```
     PROCEDURE FOO(A, B)
     BEGIN
          A = A + B
     END
```

The code generated by a simple-minded parser might be:

```
     FOO: MOVE 6(SP),D0       ; Fetch A
          MOVE D0,-(SP)       ; Push it
          MOVE 4(SP),D0       ; Fetch B
          ADD (SP)+,D0        ; Add A
          MOVE D0,6(SP)       : Store A
          RTS
```

This would be wrong. When we push the first argument onto the stack, the offsets for the two formal parameters are no longer 4 and 6, but are 6 and 8. So the second fetch would fetch A again, not B.

This is not the end of the world. I think you can see that all we really have to do is to alter the offset every time we do a push, and that in fact is what's done if the CPU has no support for other methods.

Fortunately, though, the 68000 does have such support. Recognizing that this CPU would be used a lot with high-order language compilers, Motorola decided to add direct support for this kind of thing.

The problem, as you can see, is that as the procedure executes, the stack pointer bounces up and down, and so it becomes an awkward thing to use as a reference to access the formal parameters. The solution is to define some _OTHER_ register, and use it instead. This register is typically set equal to the original stack pointer, and is called the frame pointer.

The 68000 instruction set LINK lets you declare such a frame pointer, and sets it equal to the stack pointer, all in one instruction. As a matter of fact, it does even more than that. Since this register may have been in use for something else in the calling procedure, LINK also pushes the current value of that register onto the stack. It can also add a value to the stack pointer, to make room for local variables.

The complement of LINK is UNLK, which simply restores the stack pointer and pops the old value back into the register.

Using these two instructions, the code for the previous example becomes:

```
     FOO: LINK A6,#0
          MOVE 10(A6),D0      ; Fetch A
          MOVE D0,-(SP)       ; Push it
          MOVE 8(A6),D0       ; Fetch B
          ADD (SP)+,D0        ; Add A
          MOVE D0,10(A6)      : Store A
          UNLK A6
          RTS
```

Fixing the compiler to generate this code is a lot easier than it is to explain it. All we need to do is to modify the code generation created by DoProc. Since that makes the code a little more than one line, I've created new procedures to deal with it, paralleling the Prolog and Epilog procedures called by DoMain:

```
{---------------------------------------------------------------}
{ Write the Prolog for a Procedure }

procedure ProcProlog(N: char);
begin
     PostLabel(N);
     EmitLn('LINK A6,#0');
end;


{---------------------------------------------------------------}
{ Write the Epilog for a Procedure }

procedure ProcEpilog;
begin
     EmitLn('UNLK A6');
     EmitLn('RTS');
end;
```

```
{-------------------------------------------------------------}
```

Procedure DoProc now just calls these:

```
{-------------------------------------------------------------}
{ Parse and Translate a Procedure Declaration }

procedure DoProc;
var N: char;
begin
     Match('p');
     N := GetName;
     FormalList;
     Fin;
     if InTable(N) then Duplicate(N);
     ST[N] := 'p';
     ProcProlog(N);
     BeginBlock;
     ProcEpilog;
     ClearParams;
end;
{-------------------------------------------------------------}
```

Finally, we need to change the references to SP in procedures LoadParam and StoreParam:

```
{-------------------------------------------------------------}
{ Load a Parameter to the Primary Register }

procedure LoadParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 2 * (NumParams - N);
     Emit('MOVE ');
     WriteLn(Offset, '(A6),D0');
end;


{-------------------------------------------------------------}
{ Store a Parameter from the Primary Register }

procedure StoreParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 2 * (NumParams - N);
     Emit('MOVE D0,');
     WriteLn(Offset, '(A6)');
end;
{-------------------------------------------------------------}
```

(Note that the Offset computation changes to allow for the extra push of A6.)

That's all it takes. Try this out and see how you like it.

At this point we are generating some relatively nice code for procedures and procedure calls. Within the limitation that there are no local variables (yet) and that no procedure nesting is allowed, this code is just what we need.

There is still just one little small problem remaining:

```
     WE HAVE NO WAY TO RETURN RESULTS TO THE CALLER!
```

But that, of course, is not a limitation of the code we're generating, but one inherent in the call-by-value protocol. Notice that we CAN use formal parameters in any way inside the procedure. We can calculate new values for them, use them as loop counters (if we had loops, that is!), etc. So the code is doing what it's supposed to. To get over this last problem, we need to look at the alternative protocol.

CALL-BY-REFERENCE

This one is easy, now that we have the mechanisms already in place. We only have to make a few changes to the code generation. Instead of pushing a value onto the stack, we must push an address. As it turns out, the 68000 has an instruction, PEA, that does just that.

We'll be making a new version of the test program for this. Before we do anything else,

>>>> MAKE A COPY <<<<

of the program as it now stands, because we'll be needing it again later.

Let's begin by looking at the code we'd like to see generated for the new case. Using the same example as before, we need the call

```
     FOO(X, Y)
```

to be translated to:

```
     PEA X(PC)           ; Push the address of X
     PEA Y(PC)           ; Push Y the address of Y
     BSR FOO             ; Call FOO
```

That's a simple matter of a slight change to Param:

```
{-------------------------------------------------------------}
{ Process an Actual Parameter }

procedure Param;
begin
     EmitLn('PEA ' + GetName + '(PC)');
end;
{-------------------------------------------------------------}
```

(Note that with pass-by-reference, we can't have expressions in the calling list, so Param can just read the name directly.)

At the other end, the references to the formal parameters must be given one level of indirection:

```
     FOO: LINK A6,#0
          MOVE.L 12(A6),A0    ; Fetch the address of A
          MOVE (A0),D0        ; Fetch A
```

```
          MOVE D0,-(SP)       ; Push it
          MOVE.L 8(A6),A0     ; Fetch the address of B
          MOVE (A0),D0        ; Fetch B
          ADD (SP)+,D0        ; Add A
          MOVE.L 12(A6),A0    ; Fetch the address of A
          MOVE D0,(A0)        : Store A
          UNLK A6
          RTS
```

All of this can be handled by changes to LoadParam and StoreParam:

```
{---------------------------------------------------------------}
{ Load a Parameter to the Primary Register }

procedure LoadParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 4 * (NumParams - N);
     Emit('MOVE.L ');
     WriteLn(Offset, '(A6),A0');
     EmitLn('MOVE (A0),D0');
end;


{---------------------------------------------------------------}
{ Store a Parameter from the Primary Register }

procedure StoreParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 4 * (NumParams - N);
     Emit('MOVE.L ');
     WriteLn(Offset, '(A6),A0');
     EmitLn('MOVE D0,(A0)');
end;
{---------------------------------------------------------------}
```

To get the count right, we must also change one line in ParamList:

```
     ParamList := 4 * N;
```

That should do it. Give it a try and see if it's generating reasonable-looking code. As you will see, the code is hardly optimal, since we reload the address register every time a parameter is needed. But that's consistent with our KISS approach here, of just being sure to generate code that works. We'll just make a little note here, that here's yet another candidate for optimization, and press on.

Now we've learned to process parameters using pass-by-value and pass-by-reference. In the real world, of course, we'd like to be able to deal with BOTH methods. We can't do that yet, though, because we have not yet had a session on types, and that has to come first.

If we can only have ONE method, then of course it has to be the good ol' FORTRAN method of pass-by-reference, since that's the only way procedures can ever return values to their caller.

This, in fact, will be one of the differences between TINY and KISS. In the next version of TINY, we'll use pass-by-reference

for all parameters. KISS will support both methods.

LOCAL VARIABLES

So far, we've said nothing about local variables, and our definition of procedures doesn't allow for them. Needless to say, that's a big gap in our language, and one that needs to be corrected.

Here again we are faced with a choice: Static or dynamic storage?

In those old FORTRAN programs, local variables were given static storage just like global ones. That is, each local variable got a name and allocated address, like any other variable, and was referenced by that name.

That's easy for us to do, using the allocation mechanisms already in place. Remember, though, that local variables can have the same names as global ones. We need to somehow deal with that by assigning unique names for these variables.

The characteristic of static storage, of course, is that the data survives a procedure call and return. When the procedure is called again, the data will still be there. That can be an advantage in some applications. In the FORTRAN days we used to do tricks like initialize a flag, so that you could tell when you were entering a procedure for the first time and could do any one-time initialization that needed to be done.

Of course, the same "feature" is also what makes recursion impossible with static storage. Any new call to a procedure will overwrite the data already in the local variables.

The alternative is dynamic storage, in which storage is allocated on the stack just as for passed parameters. We also have the mechanisms already for doing this. In fact, the same routines that deal with passed (by value) parameters on the stack can easily deal with local variables as well ... the code to be generated is the same. The purpose of the offset in the 68000 LINK instruction is there just for that reason: we can use it to adjust the stack pointer to make room for locals. Dynamic storage, of course, inherently supports recursion.

When I first began planning TINY, I must admit to being prejudiced in favor of static storage. That's simply because those old FORTRAN programs were pretty darned efficient ... the early FORTRAN compilers produced a quality of code that's still rarely matched by modern compilers. Even today, a given program written in FORTRAN is likely to outperform the same program written in C or Pascal, sometimes by wide margins. (Whew! Am I going to hear about THAT statement!)

I've always supposed that the reason had to do with the two main differences between FORTRAN implementations and the others: static storage and pass-by-reference. I know that dynamic storage supports recursion, but it's always seemed to me a bit peculiar to be willing to accept slower code in the 95% of cases that don't need recursion, just to get that feature when you need it. The idea is that, with static storage, you can use absolute addressing rather than indirect addressing, which should result in faster code.

More recently, though, several folks have pointed out to me that there really is no performance penalty associated with dynamic storage. With the 68000, for example, you shouldn't use absolute addressing anyway ... most operating systems require position

independent code. And the 68000 instruction

```
     MOVE 8(A6),D0
```

has exactly the same timing as

```
     MOVE X(PC),D0.
```

So I'm convinced, now, that there is no good reason NOT to use dynamic storage.

Since this use of local variables fits so well into the scheme of pass-by-value parameters, we'll use that version of the translator to illustrate it. (I _SURE_ hope you kept a copy!)

The general idea is to keep track of how many local parameters there are.   Then we use the integer in the LINK instruction to adjust the stack pointer downward to make room for them. Formal parameters are addressed as positive offsets from the frame pointer, and locals as negative offsets. With a little bit of work, the same procedures we've already created can take care of the whole thing.

Let's start by creating a new variable, Base:

```
     var Base: integer;
```

We'll use this variable, instead of NumParams, to compute stack offsets. That means changing the two references to NumParams in LoadParam and StoreParam:

```
{---------------------------------------------------------------}
{ Load a Parameter to the Primary Register }

procedure LoadParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 2 * (Base - N);
     Emit('MOVE ');
     WriteLn(Offset, '(A6),D0');
end;


{---------------------------------------------------------------}
{ Store a Parameter from the Primary Register }

procedure StoreParam(N: integer);
var Offset: integer;
begin
     Offset := 8 + 2 * (Base - N);
     Emit('MOVE D0,');
     WriteLn(Offset, '(A6)');
end;
{---------------------------------------------------------------}
```

The idea is that the value of Base will be frozen after we have processed the formal parameters, and won't increase further as the new, local variables, are inserted in the symbol table. This is taken care of at the end of FormalList:

```
{---------------------------------------------------------------}
{ Process the Formal Parameter List of a Procedure }
```

```
procedure FormalList;
begin
     Match('(');
     if Look <> ')' then begin
          FormalParam;
          while Look = ',' do begin
               Match(',');
               FormalParam;
          end;
     end;
     Match(')');
     Fin;
     Base := NumParams;
     NumParams := NumParams + 4;
end;
{--------------------------------------------------------------}
```

(We add four words to make allowances for the return address and old frame pointer, which end up between the formal parameters and the locals.)

About all we need to do next is to install the semantics for declaring local variables into the parser. The routines are very similar to Decl and TopDecls:

```
{--------------------------------------------------------------}
{ Parse and Translate a Local Data Declaration }

procedure LocDecl;
var Name: char;
begin
   Match('v');
     AddParam(GetName);
     Fin;
end;


{--------------------------------------------------------------}


{ Parse and Translate Local Declarations }

function LocDecls: integer;
var n: integer;
begin
     n := 0;
     while Look = 'v' do begin
          LocDecl;
          inc(n);
     end;
     LocDecls := n;
end;
{--------------------------------------------------------------}
```

Note that LocDecls is a FUNCTION, returning the number of locals to DoProc.

Next, we modify DoProc to use this information:

```
{--------------------------------------------------------------}
{ Parse and Translate a Procedure Declaration }

procedure DoProc;
```

```
var N: char;
      k: integer;
begin
     Match('p');
     N := GetName;
     if InTable(N) then Duplicate(N);
     ST[N] := 'p';
     FormalList;
     k := LocDecls;
     ProcProlog(N, k);
     BeginBlock;
     ProcEpilog;
     ClearParams;
end;
{---------------------------------------------------------------}
```

(I've made a couple of changes here that weren't really necessary. Aside from rearranging things a bit, I moved the call to Fin to within FormalList, and placed one inside LocDecls as well. Don't forget to put one at the end of FormalList, so that we're together here.)

Note the change in the call to ProcProlog. The new argument is the number of WORDS (not bytes) to allocate space for. Here's the new version of ProcProlog:

```
{---------------------------------------------------------------}
{ Write the Prolog for a Procedure }

procedure ProcProlog(N: char; k: integer);
begin
     PostLabel(N);
     Emit('LINK A6,#');
     WriteLn(-2 * k)
end;
{---------------------------------------------------------------}
```

That should do it. Add these changes and see how they work.

CONCLUSION

At this point you know how to compile procedure declarations and procedure calls, with parameters passed by reference and by value. You can also handle local variables. As you can see, the hard part is not in providing the mechanisms, but in deciding just which mechanisms to use. Once we make these decisions, the code to translate the constructs is really not that difficult.
I didn't show you how to deal with the combination of local parameters and pass-by-reference parameters, but that's a straightforward extension to what you've already seen. It just gets a little more messy, that's all, since we need to support both mechanisms instead of just one at a time. I'd prefer to save that one until after we've dealt with ways to handle different variable types.

That will be the next installment, which will be coming soon to a Forum near you. See you then.

```
*   Copyright (C) 1989 Jack W. Crenshaw. All rights reserved.   *
*                                                               *
*****************************************************************
```