LET'S BUILD A COMPILER!

By

Jack W. Crenshaw, Ph.D.

26 May 1990

Part XIV: TYPES

INTRODUCTION

In the last installment (Part XIII: PROCEDURES) I mentioned that in that part and this one, we would cover the two features that tend to separate the toy language from a real, usable one. We covered procedure calls in that installment. Many of you have been waiting patiently, since August '89, for me to drop the other shoe. Well, here it is.

In this installment, we'll talk about how to deal with different data types. As I did in the last segment, I will NOT incorporate these features directly into the TINY compiler at this time. Instead, I'll be using the same approach that has worked so well for us in the past: using only fragments of the parser and

single-character tokens. As usual, this allows us to get directly to the heart of the matter without having to wade through a lot of unnecessary code. Since the major problems in dealing with multiple types occur in the arithmetic operations, that's where we'll concentrate our focus.

A few words of warning: First, there are some types that I will NOT be covering in this installment. Here we will ONLY be talking about the simple, predefined types. We won't even deal with arrays, pointers or strings in this installment; I'll be covering them in the next few.

Second, we also will not discuss user-defined types. That will not come until much later, for the simple reason that I still haven't convinced myself that user-defined types belong in a language named KISS. In later installments, I do intend to cover at least the general concepts of user-defined types, records, etc., just so that the series will be complete. But whether or not they will be included as part of KISS is still an open issue. I am open to comments or suggestions on this question.

Finally, I should warn you: what we are about to do CAN add considerable extra complication to both the parser and the generated code. Handling variables of different types is straightforward enough. The complexity comes in when you add rules about conversion between types. In general, you can make the compiler as simple or as complex as you choose to make it, depending upon the way you define the type-conversion rules. Even if you decide not to allow ANY type conversions (as in Ada, for example) the problem is still there, and is built into the mathematics. When you multiply two short numbers, for example, you can get a long result.

I've approached this problem very carefully, in an attempt to Keep It Simple. But we can't avoid the complexity entirely. As has so often has happened, we end up having to trade code quality against complexity, and as usual I will tend to opt for the simplest approach.

WHAT'S COMING NEXT?

Before diving into the tutorial, I think you'd like to know where we are going from here ... especially since it's been so long since the last installment.

I have not been idle in the meantime. What I've been doing is reorganizing the compiler itself into Turbo Units. One of the problems I've encountered is that as we've covered new areas and thereby added features to the TINY compiler, it's been getting longer and longer. I realized a couple of installments back that this was causing trouble, and that's why I've gone back to using only compiler fragments for the last installment and this one. The problem is that it just seems dumb to have to reproduce the code for, say, processing boolean exclusive OR's, when the subject of the discussion is parameter passing.

The obvious way to have our cake and eat it, too, is to break up the compiler into separately compilable modules, and of course the Turbo Unit is an ideal vehicle for doing this. This allows us to hide some fairly complex code (such as the full arithmetic and boolean expression parsing) into a single unit, and just pull it in whenever it's needed. In that way, the only code I'll have to reproduce in these installments will be the code that actually relates to the issue under discussion.

I've also been toying with Turbo 5.5, which of course includes

the Borland object-oriented extensions to Pascal. I haven't decided whether to make use of these features, for two reasons. First of all, many of you who have been following this series may still not have 5.5, and I certainly don't want to force anyone to have to go out and buy a new compiler just to complete the series. Secondly, I'm not convinced that the O-O extensions have all that much value for this application. We've been having some discussions about that in CompuServe's CLM forum, and so far we've not found any compelling reason to use O-O constructs. This is another of those areas where I could use some feedback from you readers. Anyone want to vote for Turbo 5.5 and O-O?

In any case, after the next few installments in the series, the plan is to upload to you a complete set of Units, and complete functioning compilers as well. The plan, in fact, is to have THREE compilers: One for a single-character version of TINY (to use for our experiments), one for TINY and one for KISS. I've pretty much isolated the differences between TINY and KISS, which are these:

- TINY will support only two data types: The character and the 16-bit integer. I may also try to do something with strings, since without them a compiler would be pretty useless. KISS will support all the usual simple types, including arrays and even floating point.

- TINY will only have two control constructs, the IF and the WHILE. KISS will support a very rich set of constructs, including one we haven't discussed here before ... the CASE.

- KISS will support separately compilable modules.

One caveat: Since I still don't know much about 80x86 assembler language, all these compiler modules will still be written to support 68000 code. However, for the programs I plan to upload, all the code generation has been carefully encapsulated into a single unit, so that any enterprising student should be able to easily retarget to any other processor. This task is "left as an exercise for the student." I'll make an offer right here and now: For the person who provides us the first robust retarget to 80x86, I will be happy to discuss shared copyrights and royalties from the book that's upcoming.

But enough talk. Let's get on with the study of types. As I said earlier, we'll do this one as we did in the last installment: by performing experiments using single-character tokens.

THE SYMBOL TABLE

It should be apparent that, if we're going to deal with variables of different types, we're going to need someplace to record what those types are. The obvious vehicle for that is the symbol table, and we've already used it that way to distinguish, for example, between local and global variables, and between variables and procedures.

The symbol table structure for single-character tokens is particularly simple, and we've used it several times before. To deal with it, we'll steal some procedures that we've used before.

First, we need to declare the symbol table itself:

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

```
var Look: char;              { Lookahead Character }

    ST: Array['A'..'Z'] of char;   {  *** ADD THIS LINE ***}
{--------------------------------------------------------------}
```

Next, we need to make sure it's initialized as part of procedure Init:

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

procedure Init;
var i: char;
begin
   for i := 'A' to 'Z' do
      ST[i] := '?';
   GetChar;
end;
{--------------------------------------------------------------}
```

We don't really need the next procedure, but it will be helpful for debugging. All it does is to dump the contents of the symbol table:

```
{--------------------------------------------------------------}
{ Dump the Symbol Table }

procedure DumpTable;
var i: char;
begin
   for i := 'A' to 'Z' do
      WriteLn(i, ' ', ST[i]);
end;
{--------------------------------------------------------------}
```

It really doesn't matter much where you put this procedure ... I plan to cluster all the symbol table routines together, so I put mine just after the error reporting procedures.

If you're the cautious type (as I am), you might want to begin with a test program that does nothing but initializes, then dumps the table. Just to be sure that we're all on the same wavelength here, I'm reproducing the entire program below, complete with the new procedures. Note that this version includes support for white space:

```
{--------------------------------------------------------------}
program Types;

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

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

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

var Look: char;              { Lookahead Character }
```

```
    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;


{---------------------------------------------------------------}
{ Dump the Symbol Table }

procedure DumpTable;
var i: char;
begin
   for i := 'A' to 'Z' do
        WriteLn(i, ' ', ST[i]);
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;
```

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

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


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

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

OK, run this program. You should get a (very fast) printout of all the letters of the alphabet (potential identifiers), each followed by a question mark. Not very exciting, but it's a start.

Of course, in general we only want to see the types of the variables that have been defined. We can eliminate the others by modifying DumpTable with an IF test. Change the loop to read:

```
  for i := 'A' to 'Z' do
     if ST[i] <> '?' then
         WriteLn(i, ' ', ST[i]);
```

Now, run the program again. What did you get?

Well, that's even more boring than before! There was no output at all, since at this point NONE of the names have been declared. We can spice things up a bit by inserting some statements declaring some entries in the main program. Try these:

```
     ST['A'] := 'a';
     ST['P'] := 'b';
     ST['X'] := 'c';
```

This time, when you run the program, you should get an output showing that the symbol table is working right.

ADDING ENTRIES

Of course, writing to the table directly is pretty poor practice, and not one that will help us much later. What we need is a procedure to add entries to the table. At the same time, we know that we're going to need to test the table, to make sure that we aren't redeclaring a variable that's already in use (easy to do with only 26 choices!). To handle all this, enter the following new procedures:

```
{--------------------------------------------------------------}
{ Report Type of a Variable }


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


{--------------------------------------------------------------}
{ Report if a Variable is in the Table }


function InTable(N: char): boolean;
begin
   InTable := TypeOf(N) <> '?';
end;


{--------------------------------------------------------------}
{ Check for a Duplicate Variable Name }

procedure CheckDup(N: char);
begin
   if InTable(N) then Abort('Duplicate Name ' + N);
end;


{--------------------------------------------------------------}
{ Add Entry to Table }

procedure AddEntry(N, T: char);
begin
   CheckDup(N);
   ST[N] := T;
end;
{--------------------------------------------------------------}
```

Now change the three lines in the main program to read:

```
     AddEntry('A', 'a');
     AddEntry('P', 'b');
     AddEntry('X', 'c');
```

and run the program again. Did it work? Then we have the symbol table routines needed to support our work on types. In the next section, we'll actually begin to use them.

ALLOCATING STORAGE

In other programs like this one, including the TINY compiler itself, we have already addressed the issue of declaring global variables, and the code generated for them. Let's build a vestigial version of a "compiler" here, whose only function is to allow us declare variables. Remember, the syntax for a declaration is:

```
     <data decl> ::= VAR <identifier>
```

Again, we can lift a lot of the code from previous programs. The

following are stripped-down versions of those procedures. They are greatly simplified since I have eliminated niceties like variable lists and initializers. In procedure Alloc, note that the new call to AddEntry will also take care of checking for duplicate declarations:

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

procedure Alloc(N: char);
begin
   AddEntry(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 <> '.' do begin
      case Look of
        'v': Decl;
      else Abort('Unrecognized Keyword ' + Look);
      end;
      Fin;
   end;
end;
{--------------------------------------------------------------}
```

Now, in the main program, add a call to TopDecls and run the program. Try allocating a few variables, and note the resulting code generated. This is old stuff for you, so the results should look familiar. Note from the code for TopDecls that the program is ended by a terminating period.

While you're at it, try declaring two variables with the same name, and verify that the parser catches the error.

DECLARING TYPES

Allocating storage of different sizes is as easy as modifying procedure TopDecls to recognize more than one keyword. There are a number of decisions to be made here, in terms of what the syntax should be, etc., but for now I'm going to duck all the issues and simply declare by executive fiat that our syntax will be:

```
     <data decl> ::= <typename>  <identifier>
```

where:

```
     <typename> ::= BYTE | WORD | LONG
```

(By an amazing coincidence, the first letters of these names happen to be the same as the 68000 assembly code length specifications, so this choice saves us a little work.)

We can create the code to take care of these declarations with only slight modifications. In the routines below, note that I've separated the code generation parts of Alloc from the logic parts. This is in keeping with our desire to encapsulate the machine-dependent part of the compiler.

```
{---------------------------------------------------------------}
{ Generate Code for Allocation of a Variable }

procedure AllocVar(N, T: char);
begin
   WriteLn(N, ':', TAB, 'DC.', T, ' 0');
end;


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

procedure Alloc(N, T: char);
begin
   AddEntry(N, T);
   AllocVar(N, T);
end;


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

procedure Decl;
var Typ: char;
begin
   Typ := GetName;
   Alloc(GetName, Typ);
end;


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

procedure TopDecls;
begin
   while Look <> '.' do begin
      case Look of
        'b', 'w', 'l': Decl;
      else Abort('Unrecognized Keyword ' + Look);
      end;
      Fin;
   end;
end;
{---------------------------------------------------------------}
```

Make the changes shown to these procedures, and give the thing a try. Use the single characters 'b', 'w', and 'l' for the keywords (they must be lower case, for now). You will see that in each case, we are allocating the proper storage size. Note

from the dumped symbol table that the sizes are also recorded for later use. What later use? Well, that's the subject of the rest of this installment.

ASSIGNMENTS

Now that we can declare variables of different sizes, it stands to reason that we ought to be able to do something with them. For our first trick, let's just try loading them into our working register, D0. It makes sense to use the same idea we used for Alloc; that is, make a load procedure that can load more than one size. We also want to continue to encapsulate the machine-dependent stuff. The load procedure looks like this:

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

procedure LoadVar(Name, Typ: char);
begin
   Move(Typ, Name + '(PC)', 'D0');
end;
{---------------------------------------------------------------}
```

On the 68000, at least, it happens that many instructions turn out to be MOVE's. It turns out to be useful to create a separate code generator just for these instructions, and then call it as needed:

```
{---------------------------------------------------------------}
{ Generate a Move Instruction }

procedure Move(Size: char; Source, Dest: String);
begin
   EmitLn('MOVE.' + Size + ' ' + Source + ',' + Dest);
end;
{---------------------------------------------------------------}
```

Note that these two routines are strictly code generators; they have no error-checking or other logic. To complete the picture, we need one more layer of software that provides these functions.

First of all, we need to make sure that the type we are dealing with is a loadable type. This sounds like a job for another recognizer:

```
{---------------------------------------------------------------}
{ Recognize a Legal Variable Type }

function IsVarType(c: char): boolean;
begin
   IsVarType := c in ['B', 'W', 'L'];
end;
{---------------------------------------------------------------}
```

Next, it would be nice to have a routine that will fetch the type of a variable from the symbol table, while checking it to make sure it's valid:

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

```
{ Get a Variable Type from the Symbol Table }

function VarType(Name: char): char;
var Typ: char;
begin
   Typ := TypeOf(Name);
   if not IsVarType(Typ) then Abort('Identifier ' + Name +
                                        ' is not a variable');
   VarType := Typ;
end;
{---------------------------------------------------------------}
```

Armed with these tools, a procedure to cause a variable to be loaded becomes trivial:

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

procedure Load(Name: char);
begin
     LoadVar(Name, VarType(Name));
end;
{---------------------------------------------------------------}
```

(NOTE to the concerned: I know, I know, all this is all very inefficient. In a production program, we probably would take steps to avoid such deep nesting of procedure calls. Don't worry about it. This is an EXERCISE, remember? It's more important to get it right and understand it, than it is to make it get the wrong answer, quickly. If you get your compiler completed and find that you're unhappy with the speed, feel free to come back and hack the code to speed it up!)

It would be a good idea to test the program at this point. Since we don't have a procedure for dealing with assignments yet, I just added the lines:

```
     Load('A');
     Load('B');
     Load('C');
     Load('X');
```

to the main program. Thus, after the declaration section is complete, they will be executed to generate code for the loads. You can play around with this, and try different combinations of declarations to see how the errors are handled.

I'm sure you won't be surprised to learn that storing variables is a lot like loading them. The necessary procedures are shown next:

```
{---------------------------------------------------------------}
{ Store Primary to Variable }

procedure StoreVar(Name, Typ: char);
begin
   EmitLn('LEA ' + Name + '(PC),A0');
   Move(Typ, 'D0', '(A0)');
end;
```

```
{-------------------------------------------------------------}
{ Store a Variable from the Primary Register }

procedure Store(Name: char);
begin
   StoreVar(Name, VarType(Name));
end;
{-------------------------------------------------------------}
```

You can test this one the same way as the loads.

Now, of course, it's a RATHER small step to use these to handle assignment statements. What we'll do is to create a special version of procedure Block that supports only assignment statements, and also a special version of Expression that only supports single variables as legal expressions. Here they are:

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

procedure Expression;
var Name: char;
begin
   Load(GetName);
end;


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

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


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

procedure Block;
begin
   while Look <> '.' do begin
      Assignment;
      Fin;
   end;
end;
{-------------------------------------------------------------}
```

(It's worth noting that, if anything, the new procedures that permit us to manipulate types are, if anything, even simpler and cleaner than what we've seen before. This is mostly thanks to our efforts to encapsulate the code generator procedures.)

There is one small, nagging problem. Before, we used the Pascal terminating period to get us out of procedure TopDecls. This is now the wrong character ... it's used to terminate Block. In previous programs, we've used the BEGIN symbol (abbreviated 'b') to get us out. But that is now used as a type symbol.

The solution, while somewhat of a kludge, is easy enough. We'll

use an UPPER CASE 'B' to stand for the BEGIN. So change the character in the WHILE loop within TopDecls, from '.' to 'B', and everything will be fine.

Now, we can complete the task by changing the main program to read:

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

begin
   Init;
   TopDecls;
   Match('B');
   Fin;
   Block;
   DumpTable;
end.
{--------------------------------------------------------------}
```

(Note that I've had to sprinkle a few calls to Fin around to get us out of Newline troubles.)

OK, run this program. Try the input:

```
     ba        { byte a }   *** DON'T TYPE THE COMMENTS!!! ***
     wb        { word b }
     lc        { long c }
     B         { begin  }
     a=a
     a=b
     a=c
     b=a
     b=b
     b=c
     c=a
     c=b
     c=c
     .
```

For each declaration, you should get code generated that allocates storage. For each assignment, you should get code that loads a variable of the correct size, and stores one, also of the correct size.

There's only one small little problem: The generated code is WRONG!

Look at the code for a=c above. The code is:

```
     MOVE.L    C(PC),D0
     LEA       A(PC),A0
     MOVE.B    D0,(A0)
```

This code is correct. It will cause the lower eight bits of C to be stored into A, which is a reasonable behavior. It's about all we can expect to happen.

But now, look at the opposite case. For c=a, the code generated is:

```
     MOVE.B A(PC),D0
     LEA  C(PC),A0
     MOVE.L D0,(A0)
```

This is NOT correct. It will cause the byte variable A to be stored into the lower eight bits of D0. According to the rules for the 68000 processor, the upper 24 bits are unchanged. This means that when we store the entire 32 bits into C, whatever garbage that was in those high bits will also get stored. Not good.

So what we have run into here, early on, is the issue of TYPE CONVERSION, or COERCION.

Before we do anything with variables of different types, even if it's just to copy them, we have to face up to the issue. It is not the most easy part of a compiler. Most of the bugs I have seen in production compilers have had to do with errors in type conversion for some obscure combination of arguments. As usual, there is a tradeoff between compiler complexity and the potential quality of the generated code, and as usual, we will take the path that keeps the compiler simple. I think you'll find that, with this approach, we can keep the potential complexity in check rather nicely.

THE COWARD'S WAY OUT

Before we get into the details (and potential complexity) of type conversion, I'd like you to see that there is one super-simple way to solve the problem: simply promote every variable to a long integer when we load it!

This takes the addition of only one line to LoadVar, although if we are not going to COMPLETELY ignore efficiency, it should be guarded by an IF test. Here is the modified version:

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

procedure LoadVar(Name, Typ: char);
begin
   if Typ <> 'L' then
      EmitLn('CLR.L D0');
   Move(Typ, Name + '(PC)', 'D0');
end;
{-----------------------------------------------------------------}
```

(Note that StoreVar needs no similar change.)

If you run some tests with this new version, you will find that everything works correctly now, albeit sometimes inefficiently. For example, consider the case a=b (for the same declarations shown above). Now the generated code turns out to be:

```
     CLR.L D0
     MOVE.W B(PC),D0
     LEA  A(PC),A0
     MOVE.B D0,(A0)
```

In this case, the CLR turns out not to be necessary, since the

result is going into a byte-sized variable. With a little bit of work, we can do better. Still, this is not bad, and it typical of the kinds of inefficiencies that we've seen before in simple-minded compilers.

I should point out that, by setting the high bits to zero, we are in effect treating the numbers as UNSIGNED integers. If we want to treat them as signed ones instead (the more likely case) we should do a sign extension after the load, instead of a clear before it. Just to tie this part of the discussion up with a nice, red ribbon, let's change LoadVar as shown below:

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

procedure LoadVar(Name, Typ: char);
begin
   if Typ = 'B' then
      EmitLn('CLR.L D0');
   Move(Typ, Name + '(PC)', 'D0');
   if Typ = 'W' then
      EmitLn('EXT.L D0');
end;
{-------------------------------------------------------------}
```

With this version, a byte is treated as unsigned (as in Pascal and C), while a word is treated as signed.

A MORE REASONABLE SOLUTION

As we've seen, promoting every variable to long while it's in memory solves the problem, but it can hardly be called efficient, and probably wouldn't be acceptable even for those of us who claim be unconcerned about efficiency. It will mean that all arithmetic operations will be done to 32-bit accuracy, which will DOUBLE the run time for most operations, and make it even worse for multiplication and division. For those operations, we would need to call subroutines to do them, even if the data were byte or word types. The whole thing is sort of a cop-out, too, since it ducks all the real issues.

OK, so that solution's no good. Is there still a relatively easy way to get data conversion? Can we still Keep It Simple?

Yes, indeed. All we have to do is to make the conversion at the other end ... that is, we convert on the way _OUT_, when the data is stored, rather than on the way in.

But, remember, the storage part of the assignment is pretty much independent of the data load, which is taken care of by procedure Expression. In general the expression may be arbitrarily complex, so how can procedure Assignment know what type of data is left in register D0?

Again, the answer is simple: We'll just _ASK_ procedure Expression! The answer can be returned as a function value.

All of this requires several procedures to be modified, but the mods, like the method, are quite simple. First of all, since we aren't requiring LoadVar to do all the work of conversion, let's go back to the simple version:

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

```
{ Load a Variable to Primary Register }

procedure LoadVar(Name, Typ: char);
begin
   Move(Typ, Name + '(PC)', 'D0');
end;
{---------------------------------------------------------------}
```

Next, let's add a new procedure that will convert from one type to another:

```
{---------------------------------------------------------------}
{ Convert a Data Item from One Type to Another }


procedure Convert(Source, Dest: char);
begin
   if Source <> Dest then begin
      if Source  = 'B' then
         EmitLn('AND.W #$FF,D0');
      if Dest = 'L' then
         EmitLn('EXT.L D0');
   end;
end;
{---------------------------------------------------------------}
```

Next, we need to do the logic required to load and store a variable of any type. Here are the routines for that:

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

function Load(Name: char): char;
var Typ : char;
begin
   Typ := VarType(Name);
   LoadVar(Name, Typ);
   Load := Typ;
end;


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

procedure Store(Name, T1: char);
var T2: char;
begin
   T2 := VarType(Name);
   Convert(T1, T2);
   StoreVar(Name, T2);
end;
{---------------------------------------------------------------}
```

Note that Load is a function, which not only emits the code for a load, but also returns the variable type. In this way, we always know what type of data we are dealing with. When we execute a Store, we pass it the current type of the variable in D0. Since Store also knows the type of the destination variable, it can convert as necessary.

Armed with all these new routines, the implementation of our rudimentary assignment statement is essentially trivial.

Procedure Expression now becomes a function, which returns its type to procedure Assignment:

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

function Expression: char;
begin
   Expression := Load(GetName);
end;


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

procedure Assignment;
var Name: char;
begin
   Name := GetName;
   Match('=');
   Store(Name, Expression);
end;
{-------------------------------------------------------------}
```

Again, note how incredibly simple these two routines are. We've encapsulated all the type logic into Load and Store, and the trick of passing the type around makes the rest of the work extremely easy. Of course, all of this is for our special, trivial case of Expression. Naturally, for the general case it will have to get more complex. But you're looking now at the FINAL version of procedure Assignment!

All this seems like a very simple and clean solution, and it is indeed. Compile this program and run the same test cases as before. You will see that all types of data are converted properly, and there are few if any wasted instructions. Only the byte-to-long conversion uses two instructions where one would do, and we could easily modify Convert to handle this case, too.

Although we haven't considered unsigned variables in this case, I think you can see that we could easily fix up procedure Convert to deal with these types as well. This is "left as an exercise for the student."

LITERAL ARGUMENTS

Sharp-eyed readers might have noticed, though, that we don't even have a proper form of a simple factor yet, because we don't allow for loading literal constants, only variables. Let's fix that now.

To begin with, we'll need a GetNum function. We've seen several versions of this, some returning only a single character, some a string, and some an integer. The one needed here will return a LongInt, so that it can handle anything we throw at it. Note that no type information is returned here: GetNum doesn't concern itself with how the number will be used:

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

function GetNum: LongInt;
var Val: LongInt;
begin
```

```
   if not IsDigit(Look) then Expected('Integer');
   Val := 0;
   while IsDigit(Look) do begin
      Val := 10 * Val + Ord(Look) - Ord('0');
      GetChar;
   end;
   GetNum := Val;
   SkipWhite;
end;
{--------------------------------------------------------------}
```

Now, when dealing with literal data, we have one little small problem. With variables, we know what type things should be because they've been declared to be that type. We have no such type information for literals. When the programmer says, "-1," does that mean a byte, word, or longword version? We have no clue. The obvious thing to do would be to use the largest type possible, i.e. a longword. But that's a bad idea, because when we get to more complex expressions, we'll find that it will cause every expression involving literals to be promoted to long, as well.

A better approach is to select a type based upon the value of the literal, as shown next:

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

function LoadNum(N: LongInt): char;
var Typ : char;
begin
   if abs(N) <= 127 then
      Typ := 'B'
   else if abs(N) <= 32767 then
      Typ := 'W'
   else Typ := 'L';
   LoadConst(N, Typ);
   LoadNum := Typ;
end;
{--------------------------------------------------------------}
```

(I know, I know, the number base isn't really symmetric. You can store -128 in a single byte, and -32768 in a word. But that's easily fixed, and not worth the time or the added complexity to fool with it here. It's the thought that counts.)

Note that LoadNum calls a new version of the code generator routine LoadConst, which has an added argument to define the type:

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

procedure LoadConst(N: LongInt; Typ: char);
var temp:string;
begin
   Str(N, temp);
   Move(Typ, '#' + temp, 'D0');
end;
{--------------------------------------------------------------}
```

Now we can modify procedure Expression to accomodate the two

possible kinds of factors:

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

function Expression: char;
begin
   if IsAlpha(Look) then
      Expression := Load(GetName)
   else
      Expression := LoadNum(GetNum);
end;
{--------------------------------------------------------------}
```

(Wow, that sure didn't hurt too bad! Just a few extra lines do the job.)

OK, compile this code into your program and give it a try. You'll see that it now works for either variables or constants as valid expressions.

ADDITIVE EXPRESSIONS

If you've been following this series from the beginning, I'm sure you know what's coming next: We'll expand the form for an expression to handle first additive expressions, then multiplicative, then general expressions with parentheses.

The nice part is that we already have a pattern for dealing with these more complex expressions. All we have to do is to make sure that all the procedures called by Expression (Term, Factor, etc.) always return a type identifier. If we do that, the program structure gets changed hardly at all.

The first step is easy: We can rename our existing function Expression to Term, as we've done so many times before, and create the new version of Expression:

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

function Expression: char;
var Typ: char;
begin
   if IsAddop(Look) then
      Typ := Unop
   else
      Typ := Term;
   while IsAddop(Look) do begin
      Push(Typ);
      case Look of
       '+': Typ := Add(Typ);
       '-': Typ := Subtract(Typ);
      end;
   end;
   Expression := Typ;
end;
{--------------------------------------------------------------}
```

Note in this routine how each procedure call has become a function call, and how the local variable Typ gets updated at each pass.

Note also the new call to a function Unop, which lets us deal with a leading unary minus. This change is not necessary ... we could still use a form more like what we've done before. I've chosen to introduce UnOp as a separate routine because it will make it easier, later, to produce somewhat better code than we've been doing. In other words, I'm looking ahead to optimization issues.

For this version, though, we'll retain the same dumb old code, which makes the new routine trivial:

```
{---------------------------------------------------------------}
{ Process a Term with Leading Unary Operator }

function Unop: char;
begin
   Clear;
   Unop := 'W';
end;
{---------------------------------------------------------------}
```

Procedure Push is a code-generator routine, and now has a type argument:

```
{---------------------------------------------------------------}
{ Push Primary onto Stack }

procedure Push(Size: char);
begin
   Move(Size, 'D0', '-(SP)');
end;
{---------------------------------------------------------------}
```

Now, let's take a look at functions Add and Subtract. In the older versions of these routines, we let them call code generator routines PopAdd and PopSub. We'll continue to do that, which makes the functions themselves extremely simple:

```
{---------------------------------------------------------------}
{ Recognize and Translate an Add }

function Add(T1: char): char;
begin
   Match('+');
   Add := PopAdd(T1, Term);
end;


{---------------------------------------------------------------}
{ Recognize and Translate a Subtract }

function Subtract(T1: char): char;
begin
   Match('-');
   Subtract := PopSub(T1, Term);
end;
{---------------------------------------------------------------}
```

The simplicity is deceptive, though, because what we've done is to defer all the logic to PopAdd and PopSub, which are no longer

just code generation routines. They must also now take care of the type conversions required.

And just what conversion is that? Simple: Both arguments must be of the same size, and the result is also of that size. The smaller of the two arguments must be "promoted" to the size of the larger one.

But this presents a bit of a problem. If the argument to be promoted is the second argument (i.e. in the primary register D0), we are in great shape. If it's not, however, we're in a fix: we can't change the size of the information that's already been pushed onto the stack.

The solution is simple but a little painful: We must abandon that lovely "pop the data and do something with it" instructions thoughtfully provided by Motorola.

The alternative is to assign a secondary register, which I've chosen to be R7. (Why not R1? Because I have later plans for the other registers.)

The first step in this new structure is to introduce a Pop procedure analogous to the Push. This procedure will always Pop the top element of the stack into D7:

```
{---------------------------------------------------------------}
{ Pop Stack into Secondary Register }

procedure Pop(Size: char);
begin
   Move(Size, '(SP)+', 'D7');
end;
{---------------------------------------------------------------}
```

The general idea is that all the "Pop-Op" routines can call this one. When this is done, we will then have both operands in registers, so we can promote whichever one we need to. To deal with this, procedure Convert needs another argument, the register name:

```
{---------------------------------------------------------------}
{ Convert a Data Item from One Type to Another }

procedure Convert(Source, Dest: char; Reg: String);
begin
   if Source <> Dest then begin
      if Source  = 'B' then
         EmitLn('AND.W #$FF,' + Reg);
      if Dest = 'L' then
         EmitLn('EXT.L ' + Reg);
   end;
end;
{---------------------------------------------------------------}
```

The next function does a conversion, but only if the current type T1 is smaller in size than the desired type T2. It is a function, returning the final type to let us know what it decided to do:

```
{---------------------------------------------------------------}
{ Promote the Size of a Register Value }
```

```
function Promote(T1, T2: char; Reg: string): char;
var Typ: char;
begin
   Typ := T1;
   if T1 <> T2 then
      if (T1 = 'B') or ((T1 = 'W') and (T2 = 'L')) then begin
         Convert(T1, T2, Reg);
         Typ := T2;
      end;
   Promote := Typ;
end;
{-------------------------------------------------------------}
```

Finally, the following function forces the two registers to be of the same type:

```
{-------------------------------------------------------------}
{ Force both Arguments to Same Type }

function SameType(T1, T2: char): char;
begin
   T1 := Promote(T1, T2, 'D7');
   SameType := Promote(T2, T1, 'D0');
end;
{-------------------------------------------------------------}
```

These new routines give us the ammunition we need to flesh out PopAdd and PopSub:

```
{-------------------------------------------------------------}
{ Generate Code to Add Primary to the Stack }

function PopAdd(T1, T2: char): char;
begin
   Pop(T1);
   T2 := SameType(T1, T2);
   GenAdd(T2);
   PopAdd := T2;
end;


{-------------------------------------------------------------}
{ Generate Code to Subtract Primary from the Stack }

function PopSub(T1, T2: char): char;
begin
   Pop(T1);
   T2 := SameType(T1, T2);
   GenSub(T2);
   PopSub := T2;
end;
{-------------------------------------------------------------}
```

After all the buildup, the final results are almost anticlimactic. Once again, you can see that the logic is quite simple. All the two routines do is to pop the top-of-stack into D7, force the two operands to be the same size, and then generate the code.

Note the new code generator routines GenAdd and GenSub. These are vestigial forms of the ORIGINAL PopAdd and PopSub. That is,

they are pure code generators, producing a register-to-register add or subtract:

```
{---------------------------------------------------------------}
{ Add Top of Stack to Primary }

procedure GenAdd(Size: char);
begin
   EmitLn('ADD.' + Size + ' D7,D0');
end;


{---------------------------------------------------------------}
{ Subtract Primary from Top of Stack }

procedure GenSub(Size: char);
begin
   EmitLn('SUB.' + Size + ' D7,D0');
   EmitLn('NEG.' + Size + ' D0');
end;
{---------------------------------------------------------------}
```

OK, I grant you: I've thrown a lot of routines at you since we last tested the code. But you have to admit that each new routine is pretty simple and transparent. If you (like me) don't like to test so many new routines at once, that's OK. You can stub out routines like Convert, Promote, and SameType, since they don't read any inputs. You won't get the correct code, of course, but things should work. Then flesh them out one at a time.

When testing the program, don't forget that you first have to declare some variables, and then start the "body" of the program with an upper-case 'B' (for BEGIN). You should find that the parser will handle any additive expressions. Once all the conversion routines are in, you should see that the correct code is generated, with type conversions inserted where necessary. Try mixing up variables of different sizes, and also literals. Make sure that everything's working properly. As usual, it's a good idea to try some erroneous expressions and see how the compiler handles them.

WHY SO MANY PROCEDURES?

At this point, you may think I've pretty much gone off the deep end in terms of deeply nested procedures. There is admittedly a lot of overhead here. But there's a method in my madness. As in the case of UnOp, I'm looking ahead to the time when we're going to want better code generation. The way the code is organized, we can achieve this without major modifications to the program. For example, in cases where the value pushed onto the stack does _NOT_ have to be converted, it's still better to use the "pop and add" instruction. If we choose to test for such cases, we can embed the extra tests into PopAdd and PopSub without changing anything else much.

MULTIPLICATIVE EXPRESSIONS

The procedure for dealing with multiplicative operators is much the same. In fact, at the first level, they are almost identical, so I'll just show them here without much fanfare. The first one is our general form for Factor, which includes parenthetical subexpressions:

```
{--------------------------------------------------------------}
{ Parse and Translate a Factor }

function Expression: char; Forward;

function Factor: char;
begin
   if Look = '(' then begin
      Match('(');
      Factor := Expression;
      Match(')');
      end
   else if IsAlpha(Look) then
      Factor := Load(GetName)
   else
      Factor := LoadNum(GetNum);
end;


{--------------------------------------------------------------}
{ Recognize and Translate a Multiply }

Function Multiply(T1: char): char;
begin
   Match('*');
   Multiply := PopMul(T1, Factor);
end;


{--------------------------------------------------------------}
{ Recognize and Translate a Divide }

function Divide(T1: char): char;
begin
   Match('/');
   DIvide := PopDiv(T1, Factor);
end;


{--------------------------------------------------------------}
{ Parse and Translate a Math Term }

function Term: char;
var Typ: char;
begin
   Typ := Factor;
   while IsMulop(Look) do begin
      Push(Typ);
      case Look of
       '*': Typ := Multiply(Typ);
       '/': Typ := Divide(Typ);
      end;
   end;
   Term := Typ;
end;
{--------------------------------------------------------------}
```

These routines parallel the additive ones almost exactly. As before, the complexity is encapsulated within PopMul and PopDiv. If you'd like to test the program before we get into that, you can build dummy versions of them, similar to PopAdd and PopSub. Again, the code won't be correct at this point, but the parser should handle expressions of arbitrary complexity.

MULTIPLICATION

Once you've convinced yourself that the parser itself is working properly, we need to figure out what it will take to generate the right code. This is where things begin to get a little sticky, because the rules are more complex.

Let's take the case of multiplication first. This operation is similar to the "addops" in that both operands should be of the same size. It differs in two important respects:

- The type of the product is typically not the same as that of the two operands. For the product of two words, we get a longword result.
- The 68000 does not support a 32 x 32 multiply, so a call to a software routine is needed. This routine will become part of the run-time library.
- It also does not support an 8 x 8 multiply, so all byte operands must be promoted to words.

The actions that we have to take are best shown in the following table:

T1 --> / T2 V	B	W	L
B	Convert D0 to W Convert D7 to W MULS Result = W	Convert D0 to W MULS Result = L	Convert D0 to L JSR MUL32 Result = L
W	Convert D7 to W MULS Result = L	Convert D7 to W MULS Result = L	Convert D0 to L JSR MUL32 Result = L
L	Convert D7 to L JSR MUL32 Result = L	Convert D7 to L JSR MUL32 Result = L	JSR MUL32 Result = L

This table shows the actions to be taken for each combination of operand types. There are three things to note: First, we assume a library routine MUL32 which performs a 32 x 32 multiply, leaving a >> 32-bit << (not 64-bit) product. If there is any overflow in the process, we choose to ignore it and return only the lower 32 bits.

Second, note that the table is symmetric ... the two operands enter in the same way. Finally, note that the product is ALWAYS a longword, except when both operands are bytes. (It's worth noting, in passing, that this means that many expressions will end up being longwords, whether we like it or not. Perhaps the idea of just promoting them all up front wasn't all that outrageous, after all!)

Now, clearly, we are going to have to generate different code for the 16-bit and 32-bit multiplies. This is best done by having separate code generator routines for the two cases:

```
{-------------------------------------------------------------}
{ Multiply Top of Stack by Primary (Word) }

procedure GenMult;
begin
   EmitLn('MULS D7,D0')
end;


{-------------------------------------------------------------}
{ Multiply Top of Stack by Primary (Long) }

procedure GenLongMult;
begin
   EmitLn('JSR MUL32');
end;
{-------------------------------------------------------------}
```

An examination of the code below for PopMul should convince you that the conditions in the table are met:

```
{-------------------------------------------------------------}
{ Generate Code to Multiply Primary by Stack }

function PopMul(T1, T2: char): char;
var T: char;
begin
   Pop(T1);
   T := SameType(T1, T2);
   Convert(T, 'W', 'D7');
   Convert(T, 'W', 'D0');
   if T = 'L' then
      GenLongMult
   else
      GenMult;
   if T = 'B' then
      PopMul := 'W'
   else
      PopMul:= 'L';
end;
{-------------------------------------------------------------}
```

As you can see, the routine starts off just like PopAdd. The two arguments are forced to the same type. The two calls to Convert take care of the case where both operands are bytes. The data themselves are promoted to words, but the routine remembers the type so as to assign the correct type to the result. Finally, we call one of the two code generator routines, and then assign the result type. Not too complicated, really.

At this point, I suggest that you go ahead and test the program.

Try all combinations of operand sizes.

DIVISION

The case of division is not nearly so symmetric. I also have some bad news for you:

All modern 16-bit CPU's support integer divide. The manufacturer's data sheet will describe this operation as a 32 x 16-bit divide, meaning that you can divide a 32-bit dividend by a 16-bit divisor. Here's the bad news:

THEY'RE LYING TO YOU!!!

If you don't believe it, try dividing any large 32-bit number (meaning that it has non-zero bits in the upper 16 bits) by the integer 1. You are guaranteed to get an overflow exception.

The problem is that the instruction really requires that the resulting quotient fit into a 16-bit result. This won't happen UNLESS the divisor is sufficiently large. When any number is divided by unity, the quotient will of course be the same as the dividend, which had better fit into a 16-bit word.

Since the beginning of time (well, computers, anyway), CPU architects have provided this little gotcha in the division circuitry. It provides a certain amount of symmetry in things, since it is sort of the inverse of the way a multiply works. But since unity is a perfectly valid (and rather common) number to use as a divisor, the division as implemented in hardware needs some help from us programmers.

The implications are as follows:

- The type of the quotient must always be the same as that of the dividend. It is independent of the divisor.
- In spite of the fact that the CPU supports a longword dividend, the hardware-provided instruction can only be trusted for byte and word dividends. For longword dividends, we need another library routine that can return a long result.

This looks like a job for another table, to summarize the required actions:

T1 --> / T2 V	B	W	L
B	Convert D0 to W Convert D7 to L DIVS Result = B	Convert D0 to W Convert D7 to L DIVS Result = W	Convert D0 to L JSR DIV32 Result = L
W	Convert D7 to L DIVS Result = B	Convert D7 to L DIVS Result = W	Convert D0 to L JSR DIV32 Result = L

```
          |                 |                 |                 |
----------------------------------------------------------------
          |                 |                 |                 |
     L    | Convert D7 to L | Convert D7 to L |                 |
          | JSR DIV32       | JSR DIV32       | JSR DIV32       |
          | Result = B      | Result = W      | Result = L      |
          |                 |                 |                 |
----------------------------------------------------------------
```

(You may wonder why it's necessary to do a 32-bit division, when the dividend is, say, only a byte in the first place. Since the number of bits in the result can only be as many as that in the dividend, why bother? The reason is that, if the divisor is a longword, and there are any high bits set in it, the result of the division must be zero. We might not get that if we only use the lower word of the divisor.)

The following code provides the correct function for PopDiv:

```
{-----------------------------------------------------------------}
{ Generate Code to Divide Stack by the Primary }

function PopDiv(T1, T2: char): char;
begin
   Pop(T1);
   Convert(T1, 'L', 'D7');
   if (T1 = 'L') or (T2 = 'L') then begin
      Convert(T2, 'L', 'D0');
      GenLongDiv;
      PopDiv := 'L';
      end
   else begin
      Convert(T2, 'W', 'D0');
      GenDiv;
      PopDiv := T1;
   end;
end;
{-----------------------------------------------------------------}
```

The two code generation procedures are:

```
{-----------------------------------------------------------------}
{ Divide Top of Stack by Primary  (Word) }

procedure GenDiv;
begin
   EmitLn('DIVS D0,D7');
   Move('W', 'D7', 'D0');
end;


{-----------------------------------------------------------------}
{ Divide Top of Stack by Primary (Long) }

procedure GenLongDiv;
begin
   EmitLn('JSR DIV32');
end;
{-----------------------------------------------------------------}
```

Note that we assume that DIV32 leaves the (longword) result in D0.

OK, install the new procedures for division. At this point you should be able to generate code for any kind of arithmetic expression. Give it a whirl!

BEGINNING TO WIND DOWN

At last, in this installment, we've learned how to deal with variables (and literals) of different types. As you can see, it hasn't been too tough. In fact, in some ways most of the code looks even more simple than it does in earlier programs. Only the multiplication and division operators require a little thinking and planning.

The main concept that made things easy was that of converting procedures such as Expression into functions that return the type of the result. Once this was done, we were able to retain the same general structure of the compiler.

I won't pretend that we've covered every single aspect of the issue. I conveniently ignored unsigned arithmetic. From what we've done, I think you can see that to include them adds no new challenges, just extra possibilities to test for.

I've also ignored the logical operators And, Or, etc. It turns out that these are pretty easy to handle. All the logical operators are bitwise operations, so they are symmetric and therefore work in the same fashion as PopAdd. There is one difference, however: if it is necessary to extend the word length for a logical variable, the extension should be done as an UNSIGNED number. Floating point numbers, again, are straightforward to handle ... just a few more procedures to be added to the run-time library, or perhaps instructions for a math chip.

Perhaps more importantly, I have also skirted the issue of type CHECKING, as opposed to conversion. In other words, we've allowed for operations between variables of all combinations of types. In general this will not be true ... certainly you don't want to add an integer, for example, to a string. Most languages also don't allow you to mix up character and integer variables.

Again, there are really no new issues to be addressed in this case. We are already checking the types of the two operands ... much of this checking gets done in procedures like SameType. It's pretty straightforward to include a call to an error handler, if the types of the two operands are incompatible.

In the general case, we can think of every single operator as being handled by a different procedure, depending upon the type of the two operands. This is straightforward, though tedious, to implement simply by implementing a jump table with the operand types as indices. In Pascal, the equivalent operation would involve nested Case statements. Some of the called procedures could then be simple error routines, while others could effect whatever kind of conversion we need. As more types are added, the number of procedures goes up by a square-law rule, but that's still not an unreasonably large number of procedures.

What we've done here is to collapse such a jump table into far fewer procedures, simply by making use of symmetry and other simplifying rules.

TO COERCE OR NOT TO COERCE

In case you haven't gotten this message yet, it sure appears that TINY and KISS will probably _NOT_ be strongly typed languages, since I've allowed for automatic mixing and conversion of just about any type. Which brings up the next issue:

Is this really what we want to do?

The answer depends on what kind of language you want, and the way you'd like it to behave. What we have not addressed is the issue of when to allow and when to deny the use of operations involving different data types. In other words, what should be the SEMANTICS of our compiler? Do we want automatic type conversion for all cases, for some cases, or not at all?

Let's pause here to think about this a bit more. To do so, it will help to look at a bit of history.

FORTRAN II supported only two simple data types: Integer and Real. It allowed implicit type conversion between real and integer types during assignment, but not within expressions. All data items (including literal constants) on the right-hand side of an assignment statement had to be of the same type. That made things pretty easy ... much simpler than what we've had to do here.

This was changed in FORTRAN IV to support "mixed-mode" arithmetic. If an expression had any real data items in it, they were all converted to reals and the expression itself was real. To round out the picture, functions were provided to explicitly convert from one type to the other, so that you could force an expression to end up as either type.

This led to two things: code that was easier to write, and code that was less efficient. That's because sloppy programmers would write expressions with simple constants like 0 and 1 in them, which the compiler would dutifully compile to convert at execution time. Still, the system worked pretty well, which would tend to indicate that implicit type conversion is a Good Thing.

C is also a weakly typed language, though it supports a larger number of types. C won't complain if you try to add a character to an integer, for example. Partly, this is helped by the C convention of promoting every char to integer when it is loaded, or passed through a parameter list. This simplifies the conversions quite a bit. In fact, in subset C compilers that don't support long or float types, we end up back where we were in our earlier, simple-minded first try: every variable has the same representation, once loaded into a register. Makes life pretty easy!

The ultimate language in the direction of automatic type conversion is PL/I. This language supports a large number of data types, and you can mix them all freely. If the implicit conversions of FORTRAN seemed good, then those of PL/I should have been Heaven, but it turned out to be more like Hell! The problem was that with so many data types, there had to be a large number of different conversions, AND a correspondingly large number of rules about how mixed operands should be converted. These rules became so complex that no one could remember what they were! A lot of the errors in PL/I programs had to do with unexpected and unwanted type conversions. Too much of a Good Thing can be bad for you!

Pascal, on the other hand, is a language which is "strongly typed," which means that in general you can't mix types, even if they differ only in _NAME_, and yet have the same base type!

Niklaus Wirth made Pascal strongly typed to help keep programmers out of trouble, and the restrictions have indeed saved many a programmer from himself, because the compiler kept him from doing something dumb. Better to find the bug in compilation rather than the debug phase. The same restrictions can also cause frustration when you really WANT to mix types, and they tend to drive an ex-C-programmer up the wall.

Even so, Pascal does permit some implicit conversions. You can assign an integer to a real value. You can also mix integer and real types in expressions of type Real. The integers will be automatically coerced to real, just as in FORTRAN (and with the same hidden cost in run-time overhead).

You can't, however, convert the other way, from real to integer, without applying an explicit conversion function, Trunc. The theory here is that, since the numerical value of a real number is necessarily going to be changed by the conversion (the fractional part will be lost), you really shouldn't do it in "secret."

In the spirit of strong typing, Pascal will not allow you to mix Char and Integer variables, without applying the explicit coercion functions Chr and Ord.

Turbo Pascal also includes the types Byte, Word, and LongInt. The first two are basically the same as unsigned integers. In Turbo, these can be freely intermixed with variables of type Integer, and Turbo will automatically handle the conversion. There are run-time checks, though, to keep you from overflowing or otherwise getting the wrong answer. Note that you still can't mix Byte and Char types, even though they are stored internally in the same representation.

The ultimate in a strongly-typed language is Ada, which allows _NO_ implicit type conversions at all, and also will not allow mixed-mode arithmetic. Jean Ichbiah's position is that conversions cost execution time, and you shouldn't be allowed to build in such cost in a hidden manner. By forcing the programmer to explicitly request a type conversion, you make it more apparent that there could be a cost involved.

I have been using another strongly-typed language, a delightful little language called Whimsical, by John Spray. Although Whimsical is intended as a systems programming language, it also requires explicit conversion EVERY time. There are NEVER any automatic conversions, even the ones supported by Pascal.

This approach does have certain advantages: The compiler never has to guess what to do: the programmer always tells it precisely what he wants. As a result, there tends to be a more nearly one-to-one correspondence between source code and compiled code, and John's compiler produces VERY tight code.

On the other hand, I sometimes find the explicit conversions to be a pain. If I want, for example, to add one to a character, or AND it with a mask, there are a lot of conversions to make. If I get it wrong, the only error message is "Types are not compatible." As it happens, John's particular implementation of the language in his compiler doesn't tell you exactly WHICH types are not compatible ... it only tells you which LINE the error is in.

I must admit that most of my errors with this compiler tend to be errors of this type, and I've spent a lot of time with the Whimsical compiler, trying to figure out just WHERE in the line I've offended it. The only real way to fix the error is to keep

trying things until something works.

So what should we do in TINY and KISS? For the first one, I have the answer: TINY will support only the types Char and Integer, and we'll use the C trick of promoting Chars to Integers internally. That means that the TINY compiler will be _MUCH_ simpler than what we've already done. Type conversion in expressions is sort of moot, since none will be required! Since longwords will not be supported, we also won't need the MUL32 and DIV32 run-time routines, nor the logic to figure out when to call them. I _LIKE_ it!

KISS, on the other hand, will support the type Long.

Should it support both signed and unsigned arithmetic? For the sake of simplicity I'd rather not. It does add quite a bit to the complexity of type conversions. Even Niklaus Wirth has eliminated unsigned (Cardinal) numbers from his new language Oberon, with the argument that 32-bit integers should be long enough for anybody, in either case.

But KISS is supposed to be a systems programming language, which means that we should be able to do whatever operations that can be done in assembler. Since the 68000 supports both flavors of integers, I guess KISS should, also. We've seen that logical operations need to be able to extend integers in an unsigned fashion, so the unsigned conversion procedures are required in any case.

CONCLUSION

That wraps up our session on type conversions. Sorry you had to wait so long for it, but hope you feel that it was worth the wait.

In the next few installments, we'll extend the simple types to include arrays and pointers, and we'll have a look at what to do about strings. That should pretty well wrap up the mainstream part of the series. After that, I'll give you the new versions of the TINY and KISS compilers, and then we'll start to look at optimization issues.

See you then.