# Stripping the stack

Maarten Pennings

august 1990

**Abstract**

This paper is about Turbo-pascal. It demonstrates how to do a global exit, i.e. an `Exit` of several stacked procedures. Some PASCAL implementations support `Goto <label>` where `label` is outside the current block, thereby eliminating the need for a global exit. Of course you never use gotos, neither do I. But that's because the only time a `Goto` comes in handy — intra block jumps — Turbo doesn't support them.

## 1 When to use it

In this section we will introduce a program that would benefit from a global exit mechanism. We will then use this example to illustrate how a global exit is normally realised. The third section discusses some elementary stack related topics needed to understand the global exit mechanism introduced in section 4. In the next section we will incorporate the global exit in our example.

The sample program that serves our needs is a small recursive descent parser. These programs heavily rely on mutual recursion thereby forming an excellent object for testing our global exit mechanism. Our sample parser accepts a small class of expressions, defined by the following context free grammar:

| | | |
|---|---|---|
| Expr | ⟶ | Factor { '`*`' Factor } |
| Factor | ⟶ | Term { '`+`' Term } |
| Term | ⟶ | '`x`' \| '`y`' |
| Term | ⟶ | '`(`' Expr '`)`' |

The above grammer is $LL(1)$ so we can immediately derive the recursive descent parser associated with it: see figure 1.

This parser works fine. If the entered string belongs to the language The `Ok` message is printed. If not, an error message indicating the error is displayed and the program aborts.

```
Program Parse1;
Procedure Expr(Var Inp:String); Forward;
Procedure Skip(Ch:Char; Var Inp:String);
  Begin {Skip}
    If Inp[1]=Ch
      Then Inp:=Copy(Inp,2,Length(Inp)-1)
      Else Begin Writeln('"',Ch,'" expected'); Halt(1) End
  End;  {Skip}
Procedure Term(Var Inp:String);
  Begin {Term}
    Case Inp[1] Of
      '(' :  Begin Skip('(',Inp); Expr(Inp); Skip(')',Inp) End;
      'x' :  Skip('x',Inp);
      'y' :  Skip('y',Inp)
      Else Begin Writeln('"(", "x" or "y" expected'); Halt(1) End
    End
  End;  {Term}
Procedure Factor(Var Inp:String);
  Begin {Factor}
    Term(Inp); While Inp[1]='+' Do Begin Skip('+',Inp); Term(Inp) End
  End;  {Factor}
Procedure Expr(Var Inp:String);
  Begin {Expr}
    Factor(Inp); While Inp[1]='*' Do Begin Skip('*',Inp); Factor(Inp) End
  End;  {Expr}
Procedure Parse(Var Inp:String);
  Begin {Parse}
    Inp:=Inp+'@'; Expr(Inp);
    If Inp<>'@' Then Begin Writeln('Illegal char'); Halt(1) End
  End;  {Parse}
Var Inp:String;
Begin {Parse1}
  Write('Type expression:  '); Readln(Inp); Parse(Inp); Writeln('Ok')
End.  {Parse1}
```

Figure 1: Abort on error

## 2 Chaining the exits

The program presented in the previous paragraph meets its informal specification. But in real-life situations, the aborting-part of the story is inacceptable. The procedure `Parse` should always return so that the program can take additional steps to handle the error. For example, one might like to print the rest of `inp` so that the user gets to know the location of the error. And then — in a real life program — the build-in editor is invoked to edit the original input.

If you have a compiler that allows intra-block jumps, you could replace the `Halt(1)` procedures with a `Goto 9999`, where `9999` labels the statement following the `Parse(Inp)` statement. Of course you would need a global variable (or var parameter) to indicate an error has occured, so that you can take appropriate action when `Parse` returns.

But then, who has a compiler that allows intra-block jumps? Don't bother to reply, I'am too addicted to Borlands products to even consider your suggestion. So we have to solve this problem with Borlands resources. You explicitly encode `Exit`s when an error is reported: see figure 2.

```
Program Parse2;
Var Error:Boolean;
Procedure Expr(Var Inp:String); Forward;
Procedure Skip(Ch:Char; Var Inp:String);
  Begin {Skip}
    If Inp[1]=Ch
      Then Inp:=Copy(Inp,2,Length(Inp)-1)
      Else Begin Writeln('"',Ch,'" expected'); Error:=True End
  End;  {Skip}
Procedure Term(Var Inp:String);
  Begin {Term}
    Case Inp[1] Of
      '(' :  Begin Skip('(',Inp); Expr(Inp); If Error Then Exit; Skip(')',Inp) End;
      'x' :  Skip('x',Inp);
      'y' :  Skip('y',Inp)
      Else  Begin Writeln('"(", "x" or "y" expected'); Error:=True End
    End
  End;  {Term}
Procedure Factor(Var Inp:String);
  Begin {Factor}
    Term(Inp); If Error Then Exit;
    While Inp[1]='+' Do Begin Skip('+',Inp); Term(Inp); If Error Then Exit End
  End;  {Factor}
Procedure Expr(Var Inp:String);
  Begin {Expr}
    Factor(Inp); If Error Then Exit;
    While Inp[1]='*' Do Begin Skip('*',Inp); Factor(Inp); If Error Then Exit End
  End;  {Expr}
Procedure Parse(Var Inp:String);
  Begin {Parse}
    Error:=False; Inp:=Inp+'@'; Expr(Inp); If Error Then Exit;
    If Inp<>'@' Then Begin Writeln('Illegal char'); Error:=True End;
  End;  {Parse}
Var Inp:String;
Begin {Parse2}
  Write('Type expression:  '); Readln(Inp); Parse(Inp);
  If Error Then Writeln(Inp) Else Writeln('Ok')
End.  {Parse2}
```

Figure 2: Chain exits on error

Normally, you would insert the `If Error Then Exit` part after each parsing-statement, i.e. after `Expr`, `Factor`, `Term` and `Skip`. But since some of these parsing statements are known not to generate an error (skipping a '`*`' when you just tested for one), I left some of these checks out.

Nevertheless, the programm is not very ellegant. Chaining the exits increases the program size, the execution time and it decreases the programs readability. Even a `Goto 9999` solution is be better in this case.

## 3 About stacks

I can image, Borland is not too eager to implement an intra-block jump. Each time a procedure is invoked, parameters, return address and the local variables as well as some overhead

information is pushed on the stack. So if your stack consists of eight nested calls and you want to drop seven of then, you are in trouble since you have no way to know what invokation uses what part of the stack. So, how many bytes have to be dropped?

If a procedure is about to be called, what gets pushed on the stack? Well, first the procedures parameters get pushed on the stack in the order the appear in. Var parameters always occupy 4 bytes of stack space since they are called by reference (a pointer uses 2 bytes for the segment and 2 for the offset). For value parameters, the story is somewhat more complicated.

In general, if a value parameter occupies 1, 2 or 4 bytes, it is pushed directly on the stack thereby occupying 2, 2 or 4 bytes[1]. Otherwise, a 4 byte pointer to the value is pushed, and the procedure itself copies the parameter into a local buffer.

Hence, an parameter of type `array[1..2] Of Char` is pushed directly on the stack (occupying 2 bytes), while one of type `array[0..511] Of Byte` is passed by reference (occupying 4 bytes) since the structure is larger than 4 bytes. For standard types as the 10 byte `Extended` type, Borland makes an exception. All standard types are always pushed directly on the stack. For a complete overview, refer to figure 3.

| | |
|---|---|
| Boolean | 2 |
| Char | 2 |
| Enumerated | 2 |
| ShortInt | 2 |
| Byte | 2 |
| Integer | 2 |
| Word | 2 |
| LongInt | 4 |
| Subrange | hostsize |
| Pointer | 4 |
| String | pointer |
| Set | pointer |
| Single | 4 |
| Real | 6 |
| Double | 8 |
| Comp | 8 |
| Extended | 10 |
| Arrays | 1, 2 or 4 or pointer |
| Records | 1, 2 or 4 or pointer |

Figure 3: Sizes of value parameters

After all parameters have been pushed, the return address is pushed. But again, there is one exception: nested procedures. When a procedure is nested, i.e. it is declared local to

[1]Since 80?86 CPU's only support word-size pushes, byte-size parameters are pushed as words. The low order byte contains the value, the high order byte is undefined and unused.

another procedure, the callers `BP` gets pushed after all parameters have been pushed and just before the call is made. This makes the variables declared local to the parent addressable to the child (nested) procedure.

After this optionally pushed word, the return address is pushed. For a near-procedure this takes only 2 bytes. For far procedures the segment must also be pushed making a total of 4 bytes.

Hence, upon entry, the stack contains, from the top downwards, the return address (optionally including a segment), an optional reference to the parent procedure (for nested procedures) and the procedures parameters. The initialisation part of the procedure (what gets done when you press F7–trace on `Begin`) then takes over the action. The general entry code has the following form:

```
PUSH BP             ;Save the current stack frame
MOV  BP,SP          ;Set up local stack frame
SUB  SP,LocalSize   ;Allocate memory for local vars and buffers
```

The parameters and local variables and buffers are made addressable via `BP` with a `MOV BP,SP` instruction. But, before that, `BP` needs to be saved — via a `PUSH BP` — so that upon exit, the previous stack frame can be restored (the previous block can be made addressable). Two bytes extra on the stack.

Next, memory is allocated for the local variables. Local variables are those declared in a `Var` block local to the procedure. A `$A+` directive word-alligns these variables thus leaving 1 byte gaps on the stack now and then. In addition to those variables explicitly declared, Turbo also reserves local buffers for doing some string and set calculations (?). The total amount of memory occupied by the local variables and buffers is called `LocalSize`, a constant determined by the compiler. The standard exit code displayed below, deallocates the entire stackframe:

```
MOV SP,BP           ;Deallocate memory of local vars and buffers
POP BP              ;Restore previous stack frame
RET ParamSize       ;return and remove parameters
```

The total amout of memory occupied by the parameters `ParamSize` — including the optional reference to the parrent procedure — is also determined by the compiler.

The contents of the stack, once a procedure is invoked, is illustrated in figure 4. Note that the stack is one word wide and that it grows downwards. Note also that the stack frames are linked.

# 4 Global exit

Let us start this section with an example. Suppose we feed the parser with the input `x+y*(x++y)*x`. The parser encounters a `"(", "x" or "y" expected` error when the stack looks like this (press Ctrl-F3–call stack):

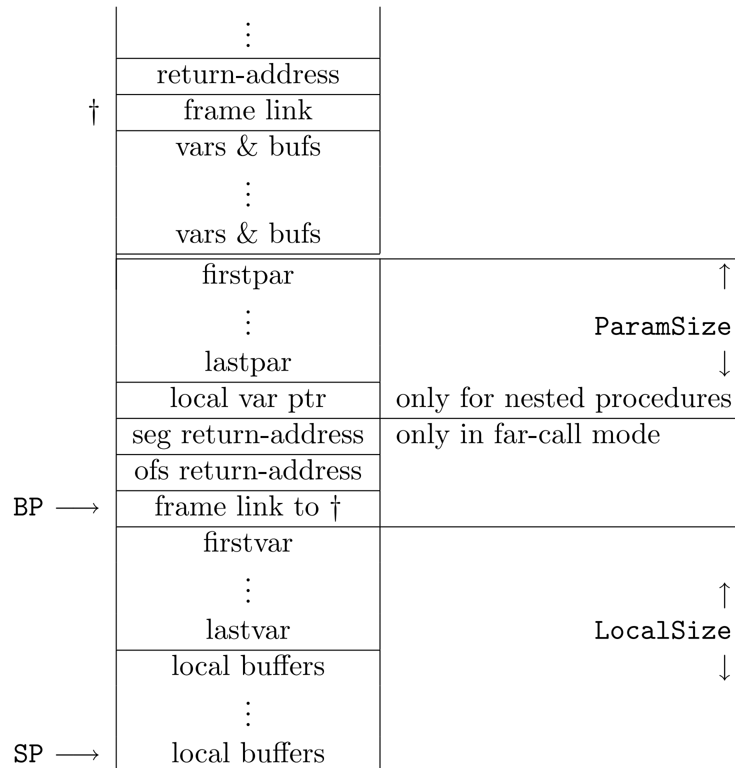


Figure 4: The stack

```
  Term('+y)*x@')
Factor('+y)*x@')
  Expr('+y)*x@')
  Term('+y)*x@')
Factor('+y)*x@')
  Expr('+y)*x@')
 Parse('+y)*x@')
Parse2()
```

If you reverse this table you get figure 5. The global exit should drop all the frames from (and including) `Parse` onwards. Therefore we must mark the stack. For this purpose we introduce a *global*, i.e. residing in the data segment, variable:

```
Var SavedSp: Word;
```

that records the stackspace just before the first call.

The global exit removes everything from `SavedSp` onwards. This is easily established by an `MOV SP,[SavedSp]` instruction. Of course we want to return to the right place also making the local variables of the caller addressable again. Therefore we do not strip the entire stack. We let `SavedSp` point to the stack frame link of the `Parse`, see figure 5 for more details.

A `POP BP` makes the local variables op `Parse2` addressable, a `RET` returns to the right spot. However a normal `RET` would leave the parameters of `Parse` on the stack, which is not what Turbo expects. We need a `RET ParamSize`.

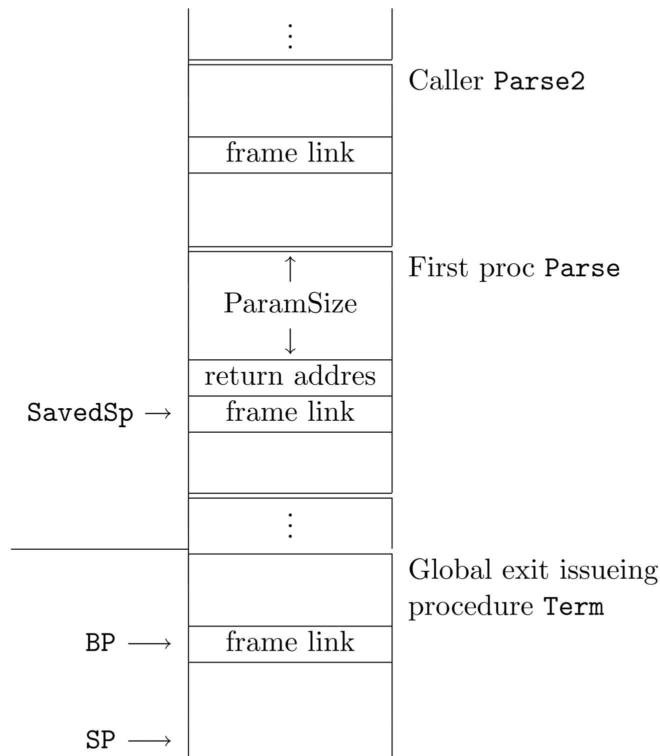


Figure 5: The stack just before a global exit

There are three pitfals here. First of all you have to know wether you far-call or near-call the first procedure, since this determines wheter you need a `RET ParamSize` or a `RETF ParamSize`.

The second problem is that you have to specify `ParamSize` for your `RET` (or `RETF`) instruction. You must count your parameters, and count them carefully. Running the standalone debugger comes in handy. Anyhow, now you know why I included the previous section and spend so much time on parameter sizes. (Don't forget to add the two bytes for a nested procedure.)

The third problem is that you have to load `SavedSp` correctly. The correct value is `Sptr-ParamSize-2-2` if the first procedure is near-call and `Sptr-ParamSize-4-2` if it is far-call. So, solving this problem boils down to solving the other two.

## 5 The final version

In our example, it is the main program that has the call that the global exit should return to. The first procedure that is called is `Parse(Var Inp:String)`. Var parameters take only four bytes of stack space. So we set `Const ParamSize=4`. We compile the `Parse` procedure with the near call model, so the global variable `Var SavedSp:Word` is set to

```
Var Inp:String;
Begin {Parse3}
  Write('Type expression: '); Readln(Inp);
  SavedSp:=SPtr-(ParamSize+2+2); Parse(Inp);
  If Error Then Writeln(Inp) Else Writeln('Ok')
End.  {Parse3}
```

All we need now is the global exit procedure. We made it an inline procedure, but if you prefer, you can make it a normal one. Keep in mind however to select the correct call model for returning: near in our example.

```
Procedure GlobalExit; Inline
  ($8B/$26/SavedSp      {MOV  SP,[SavedSp]}
  /$5D                  {POP  BP}
  /$C2/>ParamSize       {RET  ParamSize}
  );
```

Note the `>` symbol in the last instruction. It forces the generation of a word instead of a byte. If you make `Parse` a far call procedure, you must not only decrease `SavedSp` by two, you must also change `$C2` (`RET`) to `$CA` (`RETF`). For the final program see figure 6.

## 6 Once again, peace in my mind

This hacking-cyclus has consumed a considerable amout of time. I tried to get all information right, but I cannot be held responsible for that. If you get a crashing computer, first blame it on yourself. If extensive study does not solve the problem, blame it on me and let me know, I like feed-back.

Maarten Pennings
Remuslaan 3
5631 JN Eindhoven
040-461367.

```
{$F-} {Make sure to select the near call model}
Program Parse3;
Var SavedSp:Word; Const ParamSize=4;
Procedure GlobalExit; Inline($8B/$26/SavedSp/$5D/$C2/>ParamSize);
Var Error:Boolean;
Procedure Expr(Var Inp:String); Forward; {near}
Procedure Skip(Ch:Char; Var Inp:String);
  Begin {Skip}
    If Inp[1]=Ch
      Then Inp:=Copy(Inp,2,Length(Inp)-1)
      Else Begin Writeln('"',Ch,'" expected'); Error:=True; GlobalExit End
  End;  {Skip}
Procedure Term(Var Inp:String);
  Begin {Term}
    Case Inp[1] Of
      '(' :  Begin Skip('(',Inp); Expr(Inp); Skip(')',Inp) End;
      'x' :  Skip('x',Inp);
      'y' :  Skip('y',Inp)
      Else  Begin Writeln('"(", "x" or "y" expected'); Error:=True; GlobalExit End
    End
  End;  {Term}
Procedure Factor(Var Inp:String);
  Begin {Factor}
    Term(Inp); While Inp[1]='+' Do Begin Skip('+',Inp); Term(Inp) End
  End;  {Factor}
Procedure Expr(Var Inp:String); {near}
  Begin {Expr}
    Factor(Inp); While Inp[1]='*' Do Begin Skip('*',Inp); Factor(Inp) End
  End;  {Expr}
Procedure Parse(Var Inp:String);
  Begin {Parse}
    Error:=False; Inp:=Inp+'@'; Expr(Inp);
    If Inp<>'@' Then Begin Writeln('Illegal char'); Error:=True; GlobalExit End
  End;  {Parse}
Var Inp:String;
Begin {Parse3}
  Write('Type expression:  '); Readln(Inp);
  SavedSp:=SPtr-(ParamSize+2+2); {near} Parse(Inp);
  If Error Then Writeln(Inp) Else Writeln('Ok')
End.  {Parse3}
```

Figure 6: Global exit mechanism