Class 31: Stack Frames (2)

Held: Friday, 11 November 2011

Summary: Today we continue our investigation of stack frames by considering some particular examples as well as code we might write to support stack frames.

Related Pages:

- EBoard.

Notes:

- I’ve made some changes to due dates for various parts of the project.

Overview:

- Dealing with Scope, Revisited.
- Our Example, Continued.
- A Short Walkthrough.
- Implications for code generation.

Dealing with Scopes

- We know how to deal with global variables. They’re in fixed areas of memory.
- We know how to deal with variables local to a procedure. They’re at some offset from the start of the frame pointer.
- What do we do about other variables? That is, what do we do about the variables that are in enclosing scopes?
  - There are two perspectives, dynamic and static
- We’ll discuss details in class.

An Example

- Let us continue our discussion of stack frames with a simple example.
- Everyone’s favorite recursive procedure is factorial, so we’ll try that first.
- I’ve written slightly more explicit code than is normal to help you think about what is happening.

```pascal
function factorial(n: integer): integer;
var
    tmp: integer;
begin
    if (n = 0) then
        factorial := 1
    else
```
\begin{verbatim}
begin
  tmp := factorial(n-1);
  factorial := n * tmp;
end
end;
\end{verbatim}

- What will the stack frame for this procedure look like? Here’s one simple possibility (along with an indication as to where we hope to have the frame pointer and stack pointer, assuming stacks grow downward):

\begin{center}
\begin{tabular}{c|c}
\hline
used & \hline
FP & \hline
& old FP \hline
& \hline
& return value \hline
& \hline
& ret. address \hline
& \hline
SP & n \hline
& \hline
& unused \hline
\end{tabular}
\end{center}

- We’ll assume that the local procedure immediately pushes room for a local variable and a temporary, giving us the following stack.

\begin{center}
\begin{tabular}{c|c}
\hline
used & \hline
FP & \hline
& old FP \hline
& \hline
& return value \hline
& \hline
& ret. address \hline
& \hline
& n \hline
& \hline
& tmp \hline
& \hline
SP & extra \hline
& \hline
& unused \hline
\end{tabular}
\end{center}

- After a recursive call finishes, we expect to have

\begin{center}
\begin{tabular}{c|c}
\hline
used & \hline
FP & \hline
& old FP \hline
& \hline
& return value \hline
& \hline
& ret. address \hline
& \hline
& n \hline
& \hline
& tmp \hline
& \hline
SP & extra \hline
\end{tabular}
\end{center}
An Issue in Code Generation

- We’re studying this because it has an impact on where we store variables.
- In essence, we’ll need to use the symbol table to keep track of where each variable is stored.
- So, when we start a procedure declaration, we not only need to start a new scope, we also need to set up a new activation record.

```plaintext
procedure_heading :
  _PROCEDURE id
  {
    symtab_enter (symtab);
    start_new_activation_record ();
    AttributeList *attributes = new_attribute_list (3);
    set_s_attribute (attributes, "type", "procedure");
    set_p_attribute (attributes, "parameters", NULL); // filled in later
    set_p_attribute (attributes, "activation_record",
      current_activation_record ());
    symtab_put (symtab, get_s_attribute (id, "name"), attributes);
  }

parameter_list _SEMICOLON
  {
    AttributeSet *attributes = new_attribute_set (0);
    $$ = new_interior_node (_procedure_heading, attributes, 1, $2);
  }

variable_declaration :
  idlist _COLON type
  {
    AttributeSet *attributes = new_attribute_set (2);
    $$ = new_interior_node (_variable_declaration, attributes, 2, $1, $3);
    int size = type_size (type);
    Node id;
    ActivationRecord ar = current_activation_record ();
    FOR_EACH_ID (id, $1,
      {
        set_p_attribute (id->attributes, "type", type);
        set_i_attribute (id->attributes, "offset",
          allocate_variable (ar, size));
      });
  }
```

- We then want to allocate space for each variable

- Here’s a sketch of the for-each macro.
A potential complication when doing parameters: How much space do we allocate for a reference parameter?