Implementation of Scopes and Nested Blocks

– lecture 5 –

Translation Scheme

Source

```
:- Program = (
  global a ;
  a = 1 ;
  { local b,c ;
    b = a + 1 ;
    c = a + b ;
    { local d,e ;
      d = 2*a + 3 ;
      e = d - 1 ;
      b = 2*e
    } ;
    { local f,g ;
      f = b+c ;
      g = f +1 ;
      c = 2*g ;
    } ;
    a = b + c
  }
), compile(Program,'test0.s').
```

```
 .text
 .globl _entry
_entry:
 pushl %ebp
 movl %esp,%ebp
 subl $16,%esp
 movl $1,a
 movl a,%eax
 addl $1,%eax
 movl %eax,-4(%ebp)
 movl a,%eax
 addl -4(%ebp),%eax
 movl %eax,-8(%ebp)
 movl $2,%eax
 imull a,%eax
 addl $3,%eax
 movl %eax,-12(%ebp)
 movl -12(%ebp),%eax
 subl $1,%eax
 movl %eax,-16(%ebp)
 movl $2,%eax
 imull -16(%ebp),%eax
 movl %eax,-4(%ebp)
 movl -4(%ebp),%eax
 addl -8(%ebp),%eax
 movl %eax,-12(%ebp)
 movl -4(%ebp),%eax
 addl -8(%ebp),%eax
 movl %eax,-12(%ebp)
 movl -12(%ebp),%eax
 addl $1,%eax
 movl %eax,-16(%ebp)
 movl $2,%eax
 imull -16(%ebp),%eax
 movl %eax,-8(%ebp)
 movl -4(%ebp),%eax
 addl -8(%ebp),%eax
 movl %eax,a
 movl %ebp,%esp
 popl %ebp
 ret
 .data
 .globl __var_area
__var_area:
a:   .long 0
 .globl __var_name_area
__var_name_area:
a_name:   .asciz "a"
 .globl __var_ptr_area
__var_ptr_area:
a_ptr:   .long a_name
__end_var_ptr_area:
.long 0
```

Translation Scheme

Source

```
:- Program = (
  global a ;
  a = 1 ;
  { local b,c ;
    b = a + 1 ;
    c = a + b ;
    { local d,e ;
      d = 2*a + 3 ;
      e = d - 1 ;
      b = 2*e
    } ;
    { local f,g ;
      f = b+c ;
      g = f +1 ;
      c = 2*g ;
    } ;
    a = b + c
  }
), compile(Program,'test0.s').
```

Target code

```
  .text
  .globl _entry
 _entry:
  pushl %ebp
  movl %esp,%ebp
  subl $16,%esp
  movl $1,a
  movl a,%eax
  addl $1,%eax
  movl %eax,-4(%ebp)
  movl a,%eax
  addl -4(%ebp),%eax
  movl %eax,-8(%ebp)
  movl $2,%eax
  imull a,%eax
  addl $3,%eax
  movl %eax,-12(%ebp)
  movl -12(%ebp),%eax
  subl $1,%eax
  movl %eax,-16(%ebp)
  movl $2,%eax
  imull -16(%ebp),%eax
  movl %eax,-4(%ebp)
  movl -4(%ebp),%eax
  addl -8(%ebp),%eax
  movl %eax,-12(%ebp)
```

```
 movl -4(%ebp),%eax
 addl -8(%ebp),%eax
 movl %eax,-12(%ebp)
 movl -12(%ebp),%eax
 addl $1,%eax
 movl %eax,-16(%ebp)
 movl $2,%eax
 imull -16(%ebp),%eax
 movl %eax,-8(%ebp)
 movl -4(%ebp),%eax
 addl -8(%ebp),%eax
 movl %eax,a
 movl %ebp,%esp
 popl %ebp
 ret
 .data
 .globl __var_area
__var_area:
a:   .long 0
 .globl __var_name_area
__var_name_area:
a_name:  .asciz "a"
 .globl __var_ptr_area
__var_ptr_area:
a_ptr:  .long a_name
__end_var_ptr_area:
.long 0
```

Translation Scheme

- `global` introduces global variables
  - can only appear once, at the beginning of a program, outside any brace
  - defined symbols allocated into the `.data` section
- `local` introduces local variables
  - May appear right after every closing brace
  - Allocated on the stack, in the "activation frame" of the procedure
  - Accessed through the `%ebp` pointer
  - Variables in parallel scopes use same memory locations, thus saving space
- Grey and beige colored blocks represent inner blocks, and their translations
  - Variables `d` and `f` share the same location; same for `e` and `g`

Rule for global variables

```
cs((global VL ; S),Code,Ain,Aout) :-!,
                % Process global variable declarations,
                % then compile S as usual
    global_vars(VL,Ain,A0),
    cs(S,Code,A0,Aout).

% Process global variable declarations
% Each variable is added to a list stored in the attribute global_vars
% Each reference to an identifier will be checked to have been declared
% in either the global or local variable list.
global_vars((VH,VT),Ain,Aout) :-
    get_assoc(global_vars,Ain,VS,A0,[VH|VS]),
    notmember(VH,VS),!,
    global_vars(VT,A0,Aout).
global_vars(V,Ain,Aout) :-
    V =.. L, L \= [_,_|_],
    get_assoc(global_vars,Ain,VS,Aout,[V|VS]),
    notmember(V,VS),!.
```

Collect variable symbols in the attribute `global_vars`

Rule for local variables

```
local_vars_helper(V,Ain,Aout) :-
    % allocate space on the stack for a local variable
    % TopIn indexes the most recently allocated variable, so
    % the current variable will be stored at TopIn+4 (downwards from %ebp)
    % Store the maximum amount of space allocated so far in max_local_vars,
    % so as to be able to allocate space conservatively at the start of the program
    get_assoc(top_local_vars,Ain,TopIn,A0,TopOut), get_assoc(max_local_vars,A0,MaxIn,A1,MaxOut),
    get_assoc(local_vars,A1,VS,Aout,[(V,Ref)|VS]), TopOut #= TopIn + 4,
    atomic_list_concat([-,TopOut,'(%ebp)'],Ref), MaxOut #= max(MaxIn,TopOut).

% Process local variable declarations. Each variable is allocated
% on the stack, and translated into a memory reference of the form
%  -N(%ebp), where N must be a constant. Every reference to an
% identifier will be searched first in the list of local vars, and
% then in the list of global vars. For local vars, the identifier
% will be translated into the corresponding ebp-based memory reference.
local_vars((VH,VT),Ain,Aout) :- !, local_vars_helper(VH,Ain,Aaux), local_vars(VT,Aaux,Aout).
local_vars(V,Ain,Aout) :- !, V =.. L, L \= [_,_|_], local_vars_helper(V,Ain,Aout).

cs({local VL ; S},Code,Ain,Aout) :- !,
            % Preserve the original attribute, and restore at end of block
    get_assoc(local_vars,Ain,OriginalLocalVars),
    get_assoc(top_local_vars,Ain,OriginalTopLocalVars),
            % Process local variable declarations at top of block
    local_vars(VL,Ain,A0),
            % compile the rest of the statements
    cs(S,Code,A0,A1),
            % restore original list of local variables, and allocation space
    put_assoc(local_vars,A1,OriginalLocalVars,A2),
    put_assoc(top_local_vars,A2,OriginalTopLocalVars,Aout).
```

Changes in the 'if' and 'while' rules

```
cs((if B then { S }), Code,Ain,Aout) :- !,
                        % Code for 'if-then', similar to the one above.
    put_assoc(context,Ain,stmt,A1),
    comp_expr(B,CB,A1,A2),
    B =.. [Op|_],     % The condition of the jump must now be negated
    (   member((Op,I),[(<,jge),(=<,jg),(==,jne),
                       (\=,je),(>,jle),(>=,jl)])
     -> true
     ;  I = je ),
    COpB = [ '\n\t\t ',I,' ',Lifend ],
    cs({S},C,A2,A3),
    Cif = [ '\n',Lifend,':' ],
    get_assoc(labelsuffix,A3,Kin,Aout,Kout),
    generateLabels([Lifend],Kin,Kout),
    append([CB,COpB,C,Cif],Code).
```

Surrounded by braces so as to allow a new scope to be formed. Similar for the other rules for 'while' and 'if'

New 'main' predicate

```
% Main predicate
%  1st arg : Program to be compiled
%  2nd arg : File for output
% The generated file has to be compiled together with runtime-stmt.c
% to produce a valid executable. Should work on Linux, Mac, and Cygwin.
compile(P,File) :-
    tell(File),                               % open output file
    empty_assoc(Empty),                       % initialize attribute dict
    AbreakIn = Empty,
    put_assoc(break,AbreakIn,none,AbreakOut),
    AcontIn = AbreakOut,                      % initial 'break' label is none
    put_assoc(continue,AcontIn,none,AcontOut),
    AcaseendIn = AcontOut,                    % initial 'continue' label is none
    put_assoc(label_case_end,AcaseendIn,none,AcaseendOut),
    AcasetablelabelsIn = AcaseendOut,          % initial case-end label is none
    put_assoc(case_table_labels,AcasetablelabelsIn,([],[]),AcasetablelabelsOut),
    AlabelsuffixIn = AcasetablelabelsOut,      % initial table has no labels (empty lists)
    put_assoc(labelsuffix,AlabelsuffixIn,0,AlabelsuffixOut),
    AlocalvarsIn = AlabelsuffixOut,           % initialize label counter
    put_assoc(local_vars,AlocalvarsIn,[],AlocalvarsOut),
    AglobalvarsIn = AlocalvarsOut,            % initial local vars list is empty
    put_assoc(global_vars,AglobalvarsIn,[],AglobalvarsOut),
    AtoplocalIn = AglobalvarsOut,             % initial global vars list is empty
    put_assoc(top_local_vars,AtoplocalIn,0,AtoplocalOut),
    AmaxlocalIn = AtoplocalOut,               % current allocation size is 0
    put_assoc(max_local_vars,AmaxlocalIn,0,AmaxlocalOut),
                                              % max allocation size is 0
    Ainit = AmaxlocalOut,
```

New 'main' predicate

```
cs(P,Code,Ainit,Aresult),!,                 % Compile program P into Code
                                            %  -- Code is now a list of atoms
                                            %     that must be concatenated to get
                                            %     something printable
get_assoc(max_local_vars,Aresult,Max),      % Retrieve the size of storage for local vars
Pre = [ '\n\t\t .text',                     % Sandwich Code between Pre and Post
    '\n\t\t .globl _entry',                 %  -- sets up compiled code as
    '\n_entry:',                            %     procedure 'start' that can be
        '\n\t\t pushl %ebp',                %     called from runtime.c
    '\n\t\t movl %esp,%ebp',
    '\n\t\t subl $',Max,',%esp'],           % Subtract the size of storage from the stack register ;
                                            % -- thus the space between %esp and %ebp is safe to use
                                            %    even after we add procedure calls to the compiler
Post = ['\n\t\t movl %ebp,%esp',            % Code to restore the original frame pointer and return
    '\n\t\t popl %ebp',                     % to runtime-stmt.c
    '\n\t\t ret'],
append([Pre,Code,Post],All),                % The actual sandwiching happens here
atomic_list_concat(All,AllWritable),        % Now concat and get writable atom
writeln(AllWritable),                       % Print it into output file
```

New 'main' predicate

```
get_assoc(global_vars,Aresult,VarList),  % Create data declarations for all vars
allocvars(VarList,VarCode,VarNames,VarPtrs),
                                         % Code to allocate all global variables
atomic_list_concat(VarCode,WritableVars),
                                         % Compound the code into writable atom, for output into file
write('\n\t\t .data\n\t\t .globl __var_area\n__var_area:\n'),
                                         % Write declarations to output file
write(WritableVars),
                                         % Create array of strings representing
                                         % global variable names, so that vars can
                                         % be printed nicely from the runtime
atomic_list_concat(VarNames,WritableVarList),
write('\n\n\t\t .globl __var_name_area\n__var_name_area:\n'),
write(WritableVarList),
                                         % Create array of pointers to strings
                                         % so that runtime code doesn't need
                                         % to be changed every time we compile
atomic_list_concat(VarPtrs,WritableVarPtrs),
write('\n\n\t\t .globl __var_ptr_area\n__var_ptr_area:\n'),
write(WritableVarPtrs),
write('\n\n__end_var_ptr_area:\t .long 0\n'),
                                         % Put null pointer at the end of array of string pointers,
                                         % to indicate that the array has ended.
told. % close output file
```

Conclusion

- Local variables allocated on the stack
  - Prepares the ground for procedure implementation

- Parallel scopes share their storage for local variables
  - Saves memory.
  - Requires keeping track of the maximum allocated stack

- All recursive calls `cs(S)`, where `S` is a statement, must be converted into:

  `cs({S})`

  - allows a new scope to be opened, with possibly more local variables