ASM 4

Motivation

- Implement local variables
- We change the grammar a bit first

braceblock

- Previously, we had:
  braceblock $\rightarrow$ LBR stmts RBR
- Now we will have:
  braceblock $\rightarrow$ LBR var-decl-list stmts RBR

Example Syntax

```
var x number;   //global
{
    var y number;   //local
    x=10;
    if( x != 0 ){
        var x number;   //local
        x = 11;
        y = 42;
    }
}
```

Variables

- Consider: How will local variables be stored?
- We could handle locals much the way we've been dealing with globals: Allocate a chunk of RAM for each one
- Some early languages (FORTRAN) used this approach
- But: This prevents recursive functions
- Recursion is very useful, so we want to support it (when we get around to adding functions)

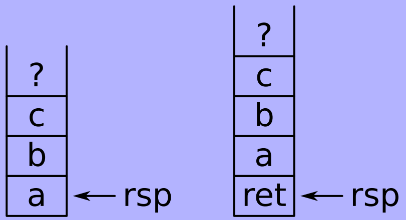
Parameters

- We'll defer the implementation of functions to later...
  - But we still need to understand how they are implemented so we can make good design decisions now
- Consider this C code:

```
void foo(int a, int b, int c){
    ...
}
```

Parameters

- Using x86 (32 bit) standard: Stack layout immediately before we call foo and immediately after:



Locals

- Modern systems use the stack to store locals as well
  - Idea: When we enter a brace block, we compute the amount of space required for locals declared in that block
  - Then we allocate space on the stack for those variables
  - When we're done with the brace block, we pop those values off the stack
- Thus, both locals and function parameters will be accessed with stack-relative addresses

Complication

- On Windows, the OS allocates stack space on demand
- If a single function uses more than ≈4KB of stack space, it needs to request space by calling __chkstk
  - It's OK to have a *total* larger than 4KB...But no single function can use more than 4KB for locals unless it calls __chkstk
- We'll ignore this detail until later
  - As long as our functions don't declare a large number of locals, we won't have a problem

Stack

- We could address variables using an offset from rsp
  - But: As we enter various brace-blocks and locals come and go, the offset from rsp for any particular variable changes
- This complicates our job
- It's possible to do, but tedious and bug-prone

Solution

- We dedicate one of the registers (rbp) as a *base pointer*
  - This always points to the *bottom* of our stack frame
    - Remember, the stack grows *down*, so the base register points to the *highest* memory address in our function's *stack frame*
    - Region of memory from base pointer to stack pointer is current function's stack frame
- When we enter the program, we need to set the base pointer to a known value

Setup

- Prologue code:

```
push rbp
mov rbp,rsp
```

- Epilogue code:

```
mov rsp,rbp
pop rbp
ret
```

Setup

- Change programNodeCode:

```
static void programNodeCode( TreeNode n ){
    //program -> var-decl-list braceblock
    ...
    emit("theRealMain:");   //existing code
    prologueCode();         //new
    vardecllistNodeCode(n.Children[0] );    //new
    braceblockNodeCode( n.Children[1] );    //existing
    epilogueCode();     //new
    emit("section .data");  //existing
    outputSymbolTableInfo();    //existing
    outputStringPoolInfo(); //existing
}
```

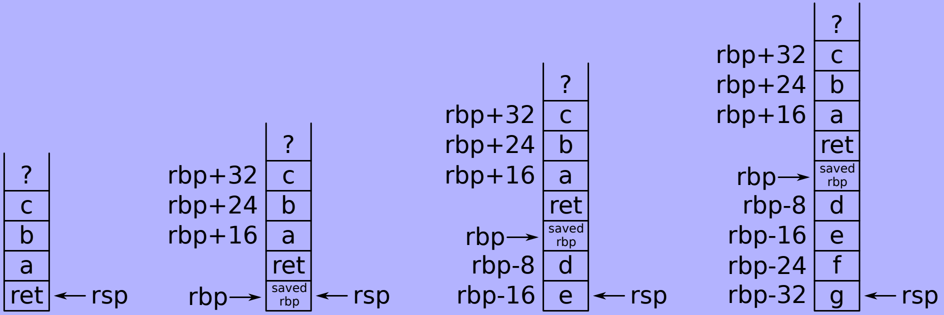
Example

- We'll now consider how locals are handled
- Consider C code like so:

```
void foo(int a, int b, int c){
    int d,e;
    ... //1
    if(...){
        int f,g;
        ... //2
    }
    ...     //3
}
```

Stack

- The stack as it exists just after the function is called but just before the prologue executes, just after the prologue executes, at point 1 (and also at point 3), and at point 2



braceblock

- Now, our new code for braceblock:

```
void braceblockNodeCode( TreeNode n ){
    //brace-block -> LBR var-decl-list stmts RBR
    vardecllistNodeCode( n.Children[1] );
    stmtsNodeCode( n.Children[2] );
}
```

- That was easy...
  - Too easy.

Variable Declarations

- We now have different variable *scopes*
- Define: A variable's scope = region of code where that variable is accessible
- We have global scope + one scope for each brace block
  - Brace block scopes are nested

Scope

- A scope is just a dictionary of variable information
  - But we'll add some functionality later, so we won't use a plain dictionary

```
class Scope{
    public Dictionary<string, VarInfo> data = new Dictionary<...>();
    public VarInfo this[string varname]{
        get {
            if( data.ContainsKey(varname) )
                return data[varname];
            else
                return null;
        }
        set {
            if( data.ContainsKey(varname) )
                error: Redeclaration
            data[varname] = value;
        }
    }
}
```

Variable Names

- What if we have variable name from outer scope re-used in inner scope?
  - Some languages (Java, C#) don't let you do this at all
  - Others (C, C++, JS with 'let') permit it: Inner scope variable *shadows* outer scope variable
  - Others (Python, JS with 'var') quietly stomp the variable from outer scope
- We'll use the C++ model
- We need to change the symbol table so it can keep track of multiple scopes

Symbol Table

- We need the following functionality:
  - Initialize
  - Get variable with a particular name from the innermost scope where it exists
  - Add new variable to innermost scope
  - Create new scope
  - Delete scope
- All of these are pretty quick to code

Symbol Table

```
class SymbolTable{
    public List<Scope> scopes = new List<Scope>();
    public SymbolTable(){
        this.AddScope();
    }
    public VarInfo this[string varname] {
        get {
            for(int i=scopes.Count-1;i>=0;i--){
                var tmp = scopes[i][varname];
                if( tmp != null )
                    return tmp;
            }
            return null;
        }
        set {
            scopes[scopes.Count-1][varname] = value;
        }
    }
    public int ScopeCount {
        get { return scopes.Count; }
    }
    public bool ContainsInCurrentScope(string varname){
        return scopes[scopes.Count-1][varname] != null;
    }
    public void AddScope(){ scopes.Add(new Scope()); }
    public void DeleteScope(){ scopes.RemoveAt(scopes.Count-1); }
}
```

Brace Block

- When we enter a brace block, we create a new scope
- When we see a variable declaration inside a brace block:
  - Allocate space on stack for the variables declared there
  - Record which stack locations correspond to which variables
- When we leave a brace block, we delete the scope and remove the variables from the stack
  - As a nice side effect, when we are ready to output globals to the data section, the symbol table will have exactly one scope in it…if we haven't made any mistakes

Braceblock

- Since brace blocks can be nested (loops, if/else), we need to know the total storage space allocated
  - Suppose block B is nested inside block A
  - Suppose blocks A and B both declare variables
  - Block B needs to know how much storage space A has allocated on the stack so it knows where its variables will be located

Inherited Attribute

- So we need to add an inherited attribute to braceblock that tells how many bytes the enclosing scopes have allocated
- And the var-decl's will also need to tell how many bytes are required for each variable
- And the var-decl-list will need to compute the total storage space required

braceblock

```
void braceblockNodeCode(TreeNode n, int
    sizeOfVariablesInEnclosingBlocks){
     //brace-block -> LBR var-decl-list stmts RBR
     symtable.AddScope();
     int sizeOfVariablesInThisBlock;
     vardecllistNodeCode(n.Children[1],
         sizeOfVariablesInEnclosingBlocks, out
         sizeOfVariablesInThisBlock);
     if(sizeOfVariablesInThisBlock>0)
          emit("sub rsp,{0}", sizeOfVariablesInThisBlock);
     stmtsNodeCode(n.Children[2], sizeOfVariablesInEnclosingBlocks
         + sizeOfVariablesInThisBlock);
     if(sizeOfVariablesInThisBlock > 0)
          emit("add rsp,{0}", sizeOfVariablesInThisBlock);
     symtable.DeleteScope();
}
```

var-decl-list

- This one is a bit tricky: It has two attributes
  - An inherited attribute that tells how much storage space has been allocated for other variables so far
  - A synthesized attribute that tells how much storage space this list requires for variables

```
void vardecllistNodeCode(TreeNode n, int sizeOfVariablesDeclaredSoFar, out int
    sizeOfVariablesInThisDeclaration){
    //var-decl-list -> var-decl SEMI var-decl-list | lambda
    if(n.Children.Count == 0) {
        sizeOfVariablesInThisDeclaration = 0;
        return;
    }
    int sz;
    vardeclNodeCode(n.Children[0], sizeOfVariablesDeclaredSoFar, out sz);
    int sz2;
    vardecllistNodeCode(n.Children[2], sizeOfVariablesDeclaredSoFar+sz, out sz2);
    sizeOfVariablesInThisDeclaration = sz + sz2;
}
```

var-decl

- Finally, we have the variable declaration itself.
- Two cases: Global (in which case there's exactly one scope) or local (we have more than one scope)
  - If global: Generate a label so the item will be added to the data section
  - If local: Generate a stack offset and use that instead

var-decl

```
void vardeclNodeCode(TreeNode n, int sizeOfVariablesDeclaredSoFar, out int
    sizeOfThisVariable){
    //var-decl -> VAR ID type
    //type -> non-array-type
    //non-array-type -> NUM | STRING
    string vname = n.Children[1].Lexeme;
    string vtypestr = n.Children[2].Children[0].Children[0].Symbol;
    VarType vtype = (VarType)Enum.Parse(typeof(VarType), vtypestr);
    if(symtable.ContainsInCurrentScope(vname)) {
        throw new CompilerException("Duplicate declaration of " + vname);
    }
    sizeOfThisVariable = 8;
    if(symtable.ScopeCount == 1) {
        //this is a global
        symtable[vname] = new VarInfo(vtype, label());
    } else {
        //the very first local is at rbp-8
        int offset = sizeOfVariablesDeclaredSoFar + 8;
        symtable[vname] = new VarInfo(vtype, "rbp-" + offset);
    }
}
```

Variable Size

- Since braceblock needs to know size of variables in enclosing blocks, we need to create inherited attributes for some nodes:
  - stmts (because of stmt)
  - stmt (because of loop and cond)
  - loop (because it creates a braceblock)
  - cond (same reason)

Return

- Don't forget to add epilogueCode() to return-stmt

Assignment

- Get the test harness working: Main.cs, GrammarData.cs, inputs.txt

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org