ASM 3

Motivation

- Work with variables
  - Also support different types
  - Primarily string and number

Grammar

- Change the grammar: We previously had:
  program $\rightarrow$ braceblock
- Now we'll have:
  program $\rightarrow$ var-decl-list braceblock

Grammar

- Add variable declarations:
  var-decl-list $\rightarrow$ var-decl SEMI var-decl-list | λ
  var-decl $\rightarrow$ VAR ID type
  type $\rightarrow$ non-array-type ~~| non-array-type LB NUM RB~~
  non-array-type $\rightarrow$ NUMBER | STRING
- Later, we might add arrays

Grammar

- Finally, add assignments:
  stmt $\rightarrow$ ... | assign SEMI
  assign $\rightarrow$ ID EQ expr

Code

```
void assignNodeCode( TreeNode n ){
    // assign -> ID EQ expr
    VarType t0;
    exprNodeCode( n.Children[2], out t0 );  //result on stack
    emit("pop rax");
    emit("mov [{0}], rax" , n.Children[0].Token.Lexeme );
}
```

- This is our first attempt
- So, of course, it's not right. What are the ways it can fail?

Problems

- Undeclared variables
- Wrong variable type (lhs and rhs type mismatch)
- Name conflict
  - What if the user does something like: rax = 42?
  - The assembler will reject the code

Solution

- Let's tackle the last problem first!
- Recall: We have a label() function which gives a unique label
- We'll always change the names that the user gives us into our own labels
- Maintain a dictionary
  - Key = variable name as specified in user's code
  - Value = Information about that variable
    - At least the label and the type
    - We might store the source code line where it was declared for error diagnostics

Symbol Table

- What we're creating is a *symbol table*
- We can create a class for this
- First, a type that holds information about a single variable

```
class VarInfo {
    public string Label;    //assembly label for this var
    public VarType VType;   //"Type" is a builtin name
    public VarInfo(VarType t, string label){
        this.VType = t;
        this.Label = label;
    }
}
```

Symbol Table

- The table itself:

```
class SymbolTable{
    public Dictionary<string, VarInfo> table = new Dictionary< ... >();
    public VarInfo this[string v]{
        get {
            return table[v];
        }
        set {
            if( table.ContainsKey(v) )
                error: Redeclaration?
            table[v] = value;
        }
    }
    public bool Contains(string v){
        return table.ContainsKey(v);
    }
}
```

- Then make a static: static SymbolTable symtable;

Code

- When code declares a global variable: We enter information about it into our symbol table

```
void vardeclNodeCode(TreeNode n){
    string vname = n.Children[1].Lexeme;
    string vtypestr = n.Children[2].Children[0].Symbol;
    VarType vtype = (VarType) Enum.Parse(typeof(VarType), vtypestr
        );
    if( symtable.Contains(vname) ){
        error: Duplicate declaration of vname
    }
    symtable[vname] = new VarInfo(vtype,label());
}
```

Assign

- Now we can go back and fix assign:

```
void assignNodeCode( TreeNode n ){
    // assign -> ID EQ expr
    var a1 = exprNodeCode( n.Children[2] );
    emit("pop rax");
    string vname = n.Children[0].Lexeme;
    if( !symtable.Contains(vname) )
        error: Undeclared variable
    if( symtable[vname].VType != a1["type"] )
        error: Type mismatch
    emit("mov [{0}], rax" , symtable[vname].Label );
}
```

- Notice that we check variable type with the synthesized type attribute that came back from expr
- Last time, we set it to always use VarType.NUMBER

Analysis

- We’ve handled undeclared variables
- And wrong variable type (lhs and rhs type mismatch)
- And name conflicts
- Life is good!
  - Right?

Problem

- But we have a bit of a problem: We don’t have any way to actually *assign* strings to variables
- Ex:
  x = “foo”
- We need to go back and tweak factor

factor

- Add two productions for factor:
  factor $\rightarrow$ NUM | LP expr RP | **STRING-CONSTANT | ID**

```
void factorNodeCode(TreeNode n, out VarType type){
    //factor -> NUM | LP expr RP | STRING-CONSTANT | ID
    var child = n.Children[0];
    switch( child.Symbol ){
        case "NUM":
            emit("push __float64__({0})",Convert.ToDouble( child.Token.Lexeme));
            type = VarType.NUMBER;
            break;
        case "LP":
            exprNodeCode( n.Children[1], type );
            break;
        case "STRING-CONSTANT":
            ???
            type = VarType.STRING;
            break;
        case "ID":
            ???
        default:
            error
    }
}
```

Decisions

- We also have to decide if we need to tweak sum and term
  - Ex: In Python, we can write things like:
    "abc" * 12
    - If we want to allow that, we'd add code to term
  - In languages like Java, C#, Python, we can do string+string
  - In Java and C# we can also do string+num
- For simplicity, we'll disallow both of these operations

Question

- How are strings to be represented? We can't realistically dump a bunch of text data on the stack
  - Too slow and inefficient!
- We'll define strings as *immutable*
- Then we'll add code for a *string pool*
  - Declare a global:
    static Dictionary<string,string> stringPool;
    - key = The string constant itself
    - value = Its label
- When we want to refer to a string, we just return its address

Note

- Observe carefully the difference:
  - mov rax, [x]
    - Moves contents of memory at label x into rax
  - mov rax, x
    - Moves the location in memory that label x refers to into rax

C

- Consider C statement: “x=y;”
  - Equivalent assembly:
    mov rax,[y]
    mov [x],rax
- Consider C statement: “x=&y;”
  - Equivalent assembly:
    mov rax,y
    mov [x],rax

Problem

- Now we can write the code for the string-constant evaluation
  - The lexeme will contain the surrounding quotation marks
  - And if the string constant had backslash escapes, those will be un-expanded
- Our string-constant node will have one synthesized attribute: The label associated with the string constant

```
void stringconstantNodeCode(TreeNode n, out string lbl){
    string s = n.Lexeme;
    s = s.Substring(1);     //leading "
    s = s.Substring(0,s.Length-1);  //trailing "
    s = s.Replace("\\n","\n");
    if( !stringPool.ContainsKey(s))
        stringPool[s] = label();
    lbl = stringPool[s] ;
}
```

factor

- Now we can write factor's case for STRING-CONSTANT

```
case "STRING-CONSTANT":
    string lbl;
    stringconstantNodeCode( child, out lbl );
    //notice: No brackets!
    emit("mov rax, {0}", lbl );
    emit("push rax");
    type = VarType.STRING ;
    break;
```

- This stores the address of the string data on the stack

factor

- We’re ready to implement the code for the “ID” case in factor
- First, make sure the identifier exists

```
case "ID":
    string vname = n.Children[0].Lexeme;
    if( !symtable.Contains(vname) )
        error: Undeclared variable
```

factor

- If the variable represents a number: The variable will contain the value itself
- If the variable represents a string: The variable contains the string's address in memory. So we just push that to the stack

```
VarInfo vi = symtable[vname];
switch(vi.VType){
    case VarType.NUMBER:
    case VarType.STRING:
        emit("mov rax,[{0}]" , symtable[vname].Label );
        emit("push rax");
        break;
    default:
        ICE
}
type =  vi.VType;
```

Done Parsing

- There's one more detail: When we're done parsing, we must output data from the symbol table and string pool:

```
emit("section .data");
outputSymbolTableInfo();
outputStringPoolInfo();
```

Symbol Table

```
void outputSymbolTableInfo(){
    foreach( var vname in symtable.table.Keys ){
        var vinfo = symtable[vname]
        if( vinfo.VType == VarType.NUMBER || vinfo.VType ==
            VarType.STRING ){
             emit( "{0}:" , vinfo.Label);
             emit( "dq 0" );
        }
    }
}
```

String Pool

```
void outputStringPoolInfo(){
    foreach( var tmp in stringPool ){
        string theString = tmp.Key;
        string lbl = tmp.Value;
        emit( lbl+":");
        byte[] B = System.Text.Encoding.ASCII.GetBytes(theString);
        foreach( byte b in B )
            emit("db {0}", b );
        emit( "db 0" ); //null terminator
    }
}
```

Assignment

- Implement the necessary code for the test harness.
  - Files: Main.cs, inputs.txt, GrammarData.cs

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