Compiling Scoped Code

- Compiling scoped code raises a set of issues because most low-level languages do not support scoping.
- Also, our low-level programming languages do not support declarations, you simply start using a variable name, therefore, all declarations need to be resolved in the source language processor.
- Declarations in the source language usually just become assignment statements in our low-level target language.
Compiling Scoped Code

- As long as all variables of the source program are declared globally there is no problem:

```
// computes the factorial of x
declare x = 3;
declare y = 1;
while (1 <= x) {
    y = y * x;
    x = x - 1;
} put y;
```

Here R$ is just a variable name prefix…it will become significant when considering scoping.
Compiling Scoped Code

Now consider the following program:

```plaintext
declare x = 1;
{
    declare x = 2;
    put x;
}
put x;
```

Expected output: 2 1

```plaintext
store R$x 1;
store R$x 2;
print R$x;
print R$x;
stop;
```

Actual output: 2 2

If we are not careful in our translation our target programs will be incorrect.
Now consider the following program:

```
declare x = 1;
{
    declare x = 2;
    put x;
}
put x;
```

In the target language we simulate scoping by adding a distinct variable prefix for each scope: R$ - global scope, R$$ first nested scope, R$$$_{}$$ second nested scope, etc.

Expected output: 2 1

Actual output: 2 1

In the target language we simulate scoping by adding a distinct variable prefix for each scope: R$ - global scope, R$$ first nested scope, R$$$_{}$$ second nested scope, etc.
Compiling Scoped Code

Now consider the following program:

```
declare x = 1;
{
    declare x = 2;
    put x;
}
{
    declare x = 3;
    put x;
}
put x;
```

```
store R$x 1;
store R$$x 2;
print R$$x;
store R$$x 3;
print R$$x;
print R$x;
stop;
```

Expected output: 2 3 1

Actual output: 2 3 1

This still works because two nested scopes at the same level can never be active at the same time.
The compiler has three phases: reader, semantic analysis (declarations), code generation.

Uses the same symbol table as the interpreter, but also computes the variable prefix.
Observations on Compilers

- Compilers do not compute values (as interpreters do)
- Compilers validate the source program, making sure that the intended behavior is correct but do not execute it
- Compilers generate code for the target machine that then executes the intended behavior
Observations on the Symbol Table

- The fact that compilers do not compute values but validate the source program has an effect on the symbol table:
  - rather than storing variable-value pairs the symbols table acts merely as a record holder for variables seen/declared
  - in our case, the symbol table stores the variable-target name pairs
Reader

options{
  output=AST;
  ASTLabelType=CommonTree;
}

prog : (stmt)+ -> ^(STMTLIST stmt+)

stmt : 'declare' VAR ('=' exp)? ';'?
     | VAR '==' exp ';';?
     | 'get' VAR ';';?
     | 'put' exp ';';?
     | 'while' '(' exp ') s='stmt
     | 'if' '(' exp ') s1='stmt ('else' s2='stmt )?
     | '{' (stmt)+ '}'

exp : relexp;

relexp : addexp (('==' ^ addexp) | ('<' ^ addexp))*;
addexp : mulexp (('+'^ mulexp) | ('-'^ mulexp))*;
mulexp : atom (('*'^ atom) | ('/'^ atom))*;
atom : ('exp ') -> exp
     | VAR
     | NUM
     | '-'v=NUM -> ^(NUM['-'+$v.text])
options{
  output=AST;  // generating a new AST with the variable names rewritten
  tokenVocab=simple2;  // use token definitions from simple2
  ASTLabelType=CommonTree;  // use CommonTree AST
  backtrack=true;  // we need backtracking because of declare initializer
}

prog : ^(STMTLIST (stmt)+)
stmt : ^('declare' VAR)
       | ^('declare' VAR exp)
       | ^('get' VAR)
       | ^('put' exp)
       | ^('while' exp stmt)
       | ^('if' exp stmt stmt?)
       |
       |
       |
       |
       |
       |
       |
       |
exp : ^('==' exp exp)
     | ^('==' exp exp)
     | ^('+'' exp exp)
     | ^('-'' exp exp)
     | ^('*'' exp exp)
     | ^('/'' exp exp)
     | VAR  -> VAR[Compile.symbolTable.lookupStaticName($VAR.text)]
     | NUM
     |
The semantics phase rewrites declaration nodes into assignment nodes!
It also rewrites all references to variables to variable names with an appropriate prefix.
stmt returns [String code]

| ("=" VAR exp) { $code = "tstore "+$VAR.text+" "+exp.code+"\n"; } |
| ("get" VAR) { System.err.println("warning: 'get' feature not implemented!"); $code = "t// get ""+$VAR.text+"\n"; } |
| ("put" exp) { $code = "tprint "+exp.code+"\n"; } |
| ("while" exp s=stmt) { |
|     String topLabel = "L"+getNewLabel(); |
|     String bottomLabel = "L"+getNewLabel(); |
|     $code = topLabel+"\n"; |
|     $code += "tjumpF "+exp.code+" "+bottomLabel+"\n"; |
|     $code += $s.code; |
|     $code += "tjump "+topLabel+"\n"; |
|     $code += bottomLabel+"\n\nnoop\n"; |
| } |
| ("if" exp s=stmt) { |
|     String endLabel = "L"+getNewLabel(); |
|     $code = "tjumpF "+exp.code+" "+endLabel+"\n"; |
|     $code += $s.code; |
|     $code += endLabel+"\n\nnoop\n"; |
| } |
| ("if" exp s1=stmt s2=stmt) { |
|     String elseLabel = "L"+getNewLabel(); |
|     String endLabel = "L"+getNewLabel(); |
|     $code = "tjumpF "+exp.code+" "+elseLabel+"\n"; |
|     $code += $s1.code; |
|     $code += "tjump "+endLabel+"\n"; |
|     $code += elseLabel+"\n"; |
|     $code += $s2.code; |
|     $code += endLabel+"\n\nnoop\n"; |
| } |
| {$code = "\n";} (BLOCKSTMT (s=stmt {$code += $s.code;})+) |

Note: No Decl statement nodes!
public class SymbolTable {

    private SymbolTableScope globalScope = new SymbolTableScope(null,"R$");
    private SymbolTableScope currentScope = globalScope;

    public SymbolTableScope pushScope() {
        String newPrefix = currentScope.getStaticScopePrefix() + "$");
        currentScope = new SymbolTableScope(currentScope,newPrefix);
        return currentScope;
    }

    public SymbolTableScope popScope() {
        // go up one entry
        currentScope = currentScope.getParentScope();
        return currentScope;
    }

    public void declareSymbol(String symbol) {
        // check that the current symbol was not already declared in the
        // current scope, if so then we have an error
        if (currentScope.lookupSymbol(symbol) != null) {
            System.err.println("Error: redclaring symbol "+symbol+".");
            System.exit(1);
        }
        // all clear...enter the symbol into the scope
        String staticName = currentScope.getStaticScopePrefix() + symbol;
        currentScope.enterSymbol(symbol,staticName);
    }

    ...
}
... public String lookupSymbol(String symbol) {
    // lookup the symbol in the current scope
    SymbolTableScope lookupScope = currentScope;
    String value = lookupScope.lookupSymbol(symbol);

    // if not in current scope search up the stack
    while (value == null) {
        lookupScope = lookupScope.getParentScope();
        if (lookupScope == null) {
            // no parent scope, symbol not found
            System.err.println("Error (lookup): symbol "+symbol+" not declared.");
            // could do some more intelligent recovery here.
            System.exit(1);
            return null;
        }
        value = lookupScope.lookupSymbol(symbol);
    }
    // all done, return the value, guaranteed to be here
    // by the nature of our search procedure
    return value;
}

public String lookupStaticName(String symbol) {
    return lookupSymbol(symbol);
}
public class SymbolTableScope {
    // scope stack is built as a linked list
    private SymbolTableScope parentScope = null;

    // symbols are kept in a hashmap indexed by their name
    private HashMap<String, String> symbols = new HashMap<>();

    // a scope prefix string for variable names
    private String staticScopePrefix = null;

    public SymbolTableScope(SymbolTableScope parentScope, String staticScopePrefix) {
        this.parentScope = parentScope;
        this.staticScopePrefix = staticScopePrefix;
    }

    public String getStaticScopePrefix() {
        return staticScopePrefix;
    }

    public SymbolTableScope getParentScope() {
        return parentScope;
    }

    public void enterSymbol(String name, String value) {
        symbols.put(name, value);
    }

    public String lookupSymbol(String name) {
        return symbols.get(name);
    }
}
Example Code

- SIMPLE2COMPILER.zip