CS453 Classes and Methods
Declarations and Calls
Symbol Tables, Structure and Creation
Type Checking
(formal) variable allocation
Code Generation
Plan for Today

PA4 Structure

Class and method Declarations

Symbol Table
  Structure
  Creation
  Use

Type Checking

Code Generation
Class and Method Declarations

In PA4 we extend Meggy-Java with
tones, less than
class and method declarations
calls (statements and expressions)
identifiers (for formal parameters)

Program ::= MainClass  (ClassDecl)*

ClassDecl ::= CLASS id “{" (MethodDecl)* “}”

MethodDecl ::= PUBLIC type id “{" comma separated type , id list “}”
           “{" stmt * ( RETURN expression )? “}”

- formal id-s can occur only once in a method

   call ::= expression DOT id “{" comma separated expression list “}” SEMI

- no sub ranges (blocks inside statements cannot have variable declarations)
- no sub classes
- method IDs can occur only once in a class (no overloading)
Abstract Syntax Tree Extensions

Program

MainClass

TopClassDecl

TopClassDecl...

MethodDecl

MethodDecl...

(return) Type

formal (type,id)

formal ...

statement

Call

(receiver object) Expression

actual ...

(actual)Expression
Type Checker Extensions

Our type Checker already associates and checks types of expressions

When encountering a call
  - Check whether the method is in scope:
    an area in a program where a certain definition is visible
    in a scope, an entity (e.g. id) is bound to its meaning (e.g. type)
    - - this is implemented with a dictionary that maps strings to
      symbol table entries.
  - Check parameters (number, types, doubly defined)
  - Establish (return) type

When encountering an id
  - Check whether it is defined in scope
  - Establish its type
What is the scope of a class?

What is the scope of a method?

What is the scope of a formal?

What is the scope of instance variables?

What is the scope of local variables?

How is this different from full blown java?
Scope vs lifetime

lifetime - how long the variable must be stored at runtime

globals               whole program execution

locals                 while each instance of function is executed

static variables        whole program execution

member (instance) variables
                         from the time object is allocated
                         until object is deallocated

Which do we have in Meggy.java?
Symbol Table

Type Checking is be done with a symbol table associating types with expressions reflecting the (tree structured) scopes in the program recording the formals and return type for each method

So in Meggy Java we really have two symbol tables:
1. The expression to types “symbol table” (in the AST)
2. The identifier to meaning symbol table
   where an identifier can be
   class, method, formal, instance (member) or a local variable

Information maintained for id-s is a tree structured environment:
- set of bindings: (entity → meaning)
  e.g. method → signature (formals + return type)
- for each identifier: type, scope (includes lifetime), visibility, and run-time location

for named scopes(classes,methods), the set of identifiers it contains.
Symbol tables, and consequently scopes, in MeggyJava PA4:

program: classes : each class defines its own scope of methods
  no sub classes
  no objects (but they will come in PA5)

class: methods : each method defines its own scope of formals (name, type)
  + return type
  no variables (but they will come in PA5)

While processing the program, the current scope is kept on a (scope)stack
SymTable and STE classes

A scope has a dictionary for all its bindings

A formal is represented as a variable, it has an offset and a base. For a formal the base is FP (or Y). Other variables have different offsets (PA5: this)
SymTable, Scope, and STE classes

while building or walking a symbol table, a scope stack keeps track of the currently open scopes
Some Symbol Table Operations

While accessing the symbol table maintain a stack of scopes with the current most deeply nested scope at the top of the stack. The symbol table also maintains a reference to the outermost (or global) scope.

void insertAndPushScope(NamedScopeSTE)

For first time a named scope is created like a MethodSTE.

void pushScope(String)

Looks up a named scope like a method and then pushes its scope on the stack.

void popScope()

Pops the top scope off the stack.

STE lookup(String)

Looks for given symbol in this scope, or if it doesn't find it, in enclosing scope.

void insert(STE)

Inserts symbol table entry into current scope.
Building the Symbol Table (BuildSymTable visitor)

BuildSymTable constructor should initialize the symbol table, e.g.
    mCurrentST = new symtable.SymTable();
inTopClass: create ClassSTE, insert it and pushScope
outTopClass: popScope

inMethod:
    create a list of formal types
    create a signature: formalTypes → return type
    create a methodSTE with class&method name, node, signature
    insert it and pushScope
    create an entry for “this”
out Method: popScope
Extending the CheckTypes visitor

In PA3, CheckTypes is used for establishing types for operands and operators (like IntLiteral) and and checking child types for operators e.g. byte, equals, plus, minus, not:

```java
public void outIntegerExp(IntLiteral node){
    this.mCurrentST.setExpType(node, Type.INT);
}

public void outNotExp(notExp node){
    if (this.mCurrentST.getExpType(node.getExp()) != Type.BOOL)
        throw new semanticException(...);
    this.mCurrentST.setExpType(node, Type.Bool);
}
```

In PA4, the symbol table is initialized by BuildSymTable and must be extended by CheckTypes for the new nodes in the AST.
CheckTypes methods for Class and Method Decls

**inTopClassDecl:**
create an instance variable of type classSTE, and give it the name of the class (used for setting the type of “this”)
mCurrentST.Class pushScope (node.getName);

**inMethodDecl(MethodDecl node):**
push a new scope:  mCurrentST.pushScope(node.getName());

**outMethodDecl(MethodDecl node):**
look up method in class scope:
MethodSTE  methodSTE =
  (MethodSTE)mCurrentST.lookup(node.getName());
check that it is not defined already in this class scope (no overloading)
check that the return type in the signature methodSTE conforms with the type of the return expression
CheckTypes methods for Id and call (callStmt, callExpr)

outIdLiteral:
  lookup the STE associated with node.getLexeme in
  check that it is in scope,
    if it is not there, throw a semanticException
  get its type IdT, and mCurrentST.setExpType(node, IdT);

outCall:
  typeCheck(receiver Expr, id, Args) and produce returnType;
  mCurrentST.setExpType(node, returnType);
type Checking a call

[typeCheck](receiver, funcName, args):
  check that receiver is of type Class

receiverInfo = lookupClass(receiverType.getClassName())
invocationInfo = receiverInfo.getScope().lookup(funcName);
if it isn’t there throw exception

generate the Signature of the invoked receiver
check argument count
for each actual argument type check that it is equal to the formal type
  (think about widening when argument passing)

the type of the call is the return type from the signature
Each function call is associated with a (run time) stack frame, containing
- a representation of **this**: a reference to the instance/member variables
- the function formal parameters
- the local variables
  - - in PA4, local variables don’t play a role yet (but they will in PA5)
- a reference to the previous stack frame
- a return address (next sequential instruction of the caller)

While the function is executing it will use the RTS for expression evaluation (think of that dynamically happening above the stack frame)

To address the local variables and function parameters we use a frame pointer, that points to the top of the stack frame. (This is called a base).

Before it can generate code, the compiler needs to determine the offset off the base for each function parameter and local variable.
- this is done in by the **allocVars** visitor.
For each parameter (and later local variable), determine its offset off the stack pointer. (The FP usually points one location below the first param, so first param’s address is \( Y+1 \)). In the Meggy-Java compiler this is done in the AVRallocVars visitor, right after type checking.

Simple example code:

```java
class PA4simple {
    public static void main(String[] whatever) {
        new C().setP((byte)3, (byte)7, Meggy.Color.BLUE);
    }
}

class C {
    public void setP(byte x, byte y, Meggy.Color c) {
        Meggy.setPixel(x, y, c);
    }
}
```
Program

MainClass

TopClassDecl

BlockStatement

MethodDecl

VoidType

Formal

Formal

Formal

MeggySetPixel

CallStatement

NewExp

ByteCast

ByteCast

ColorLiteral Meggy.Color.BLUE

ByteType

ByteType

ColorType

IdLiteral x

IdLiteral y

IdLiteral c

IntLiteral 3

IntLiteral 7
Symbol table for simple before allocVars

Scope
mDict[C]
  ClassSTE
  mName = C
  mMain = false
  mSuperClass = null
  mScope

Scope
mDict[setP]
  MethodSTE
  mName = setP
  mSignature = (BYTE, BYTE, COLOR) returns class_null;
  mScope

Scope
mDict[this]
  VarSTE
  mName = this
  mType = class_C;
  mBase = INVALID
  mOffset = 0

mDict[x]
  VarSTE
  mName = x
  mType = BYTE
  mBase = INVALID
  mOffset = 0

mDict[y]
  VarSTE
  mName = y
  mType = BYTE
  mBase = INVALID
  mOffset = 0

mDict[c]
  VarSTE
  mName = c
  mType = COLOR
  mBase = INVALID
  mOffset = 0
Symbol table for simple after allocVars

<table>
<thead>
<tr>
<th>Scope</th>
<th>mDict[C]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ClassSTE</th>
<th>mDict[C]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mName = C</td>
<td></td>
</tr>
<tr>
<td>mMain = false</td>
<td></td>
</tr>
<tr>
<td>mSuperClass = null</td>
<td></td>
</tr>
<tr>
<td>mScope</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scope</th>
<th>mDict[setP]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MethodSTE</th>
<th>mDict[setP]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mName = setP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mSignature = (BYTE, BYTE, COLOR) returns class_null;</th>
<th>mDict[setP]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>VarSTE</th>
<th>mDict[this]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mName = this</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mType = class_C</th>
<th>mDict[this]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mBase = Y</td>
<td>mDict[this]</td>
</tr>
<tr>
<td>mOffset = 1</td>
<td>mDict[this]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VarSTE</th>
<th>mDict[x]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mName = x</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mType = BYTE</th>
<th>mDict[x]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mBase = Y</td>
<td>mDict[x]</td>
</tr>
<tr>
<td>mOffset = 3</td>
<td>mDict[x]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VarSTE</th>
<th>mDict[y]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mName = y</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mType = BYTE</th>
<th>mDict[y]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mBase = Y</td>
<td>mDict[y]</td>
</tr>
<tr>
<td>mOffset = 4</td>
<td>mDict[y]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VarSTE</th>
<th>mDict[c]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mName = c</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mType = COLOR</th>
<th>mDict[c]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mBase = Y</td>
<td>mDict[c]</td>
</tr>
<tr>
<td>mOffset = 5</td>
<td>mDict[c]</td>
</tr>
</tbody>
</table>
AVRallocVars visitor

PA4: for each method, go through the formals
    keeping track of an offset

inMethodDecl: allocate space for implicit “this” parameter at offset
              increment offset with size of “this” (2: reg pair)

outFormal: allocate space for formal at offset, offset += parameter size

outMethodDecl: set the number of bytes the stack frame occupies
               (current value of offset)
Function Calls

Two sides: **Caller** and **Callee** (called function)

Function(s) are associated with **stack frames**

- We cannot keep all function data in registers (recursive functions)
- For each function call there will be a stack frame on the RTS
- AllocVars has determined the position of the formals in the stack frame

The Stack Pointer moves up and down evaluating expressions; the **Frame pointer** (FP) is a fixed point in the stack frame

- So that locals (formal parameters and (later) local variables) can be addressed as an offset from the FP

**Caller** passes actual parameters through register(pair)s

- registers used: p1: r24(25)  p2: r22(23) down to p9: r8(9)
- call: receiver.funName(actual expr 1, actual expr 2) …

**Callee** passes result back through register(s) 8-bit r24, 16 bit r25,24
this: the object the function is executing in
will become important in PA6: objects, assignments, instance variables
allocVars already takes this into account
this is the **implicit first parameter** for all function calls. Not
needed yet in PA4, but we might as well put a place holder for it.

---

**FP:**
- frame pointer
  - (base: Y)
  - used to access formals / locals
  - F2
  - F3 ...

**OFP:**
- saved old frame pointer

**SP:**
- stack pointer
  - used (push pop) for expr. eval

**frame:**
- this

**formals and locals (PA5)**
Calling convention

Caller:
1. gather actual params on the RTS
   - push receiver
     - - (receiver = “this” in callee)
   - push explicit parameters 2,3,…
2. call
   - pop actuals in reg (pair)s
   - call fname
     - - (fname = className+funcName)
3. on return
   - push return value on stack
Calling convention

Caller:
1. gather actual params on the RTS
   - push receiver
     - (receiver = “this” in callee)
   - push explicit parameters 2,3,…
2. call
   - pop actuals in reg (pair)s
   - call fname
     - (fname = className+funcName)
3. on return
   - push return value on stack

Callee:
1. push old FP (r28, r29)
2. make space for frame
   - multiple push 0-s
3. copy SP → FP
4. populate frame (Reg → Y+offset)
5. execute body
   - may push return value
6. may get return value into r24(25)
7. clear frame space (undo 2)
8. pop FP into r28,r29
9. ret
PA4simple.java example: call
new C().setP((byte)3,(byte)7,Meggy.Color.BLUE);

1. caller pushes
   actual params
   new C() = 0,0
   (byte) 3
   (byte) 7
   (byte) BLUE
   and pops them into
   r18: BLUE r20: 7
   r22: 3 r24(25) newC()

2. caller performs
   CALL CsetP

CsetP: callee

1. saves mFP on RTS
2. makes space for
   parameters by pushing 0-s
3. copies SP to FP
   callee
4. populates stack frame
5. executes body
   ( 6. in case of return pops ret expr in r24(25) )
PA4simple.java example: return

\[ \text{SP}, \text{FP} \]

0
0
3
7
BLUE
mFP
NSI

\[ \text{SP} \]
\[ \text{mFP} \]

\begin{itemize}
  \item \text{calleee}
    \begin{itemize}
      \item (0. in case of return evaluates ret expr puts it in r24(25))
      \item 1. pops frame off RTS exposing mFP
      \item 2. pops mFP into FP exposing NSI
      \item 3. executes ret, which pops NSI and jumps to it
    \end{itemize}
\end{itemize}

\begin{itemize}
  \item \text{caller}
  \begin{itemize}
    \item resumes execution at NSI:
  \end{itemize}
\end{itemize}
Write a Meggy Java program with a recursive method

/* Recursively put BLUE pixels in (2,0),(1,0),(0,0)
Complete below and think about the RTS bahavior */

import meggy.Meggy;
class RecursiveCount {
    public static void main(String[] whatever){
        new Foo().count((byte)0);
    }
}
class Foo {
    public void count(byte p) {
        // if haven’t reached 2,
        // recursively call count
        // call setPixel at (p,0)
    }
}
// We will play with it in the next lecture