Big Picture

PA2

- Syntax-directed code generation

PA3

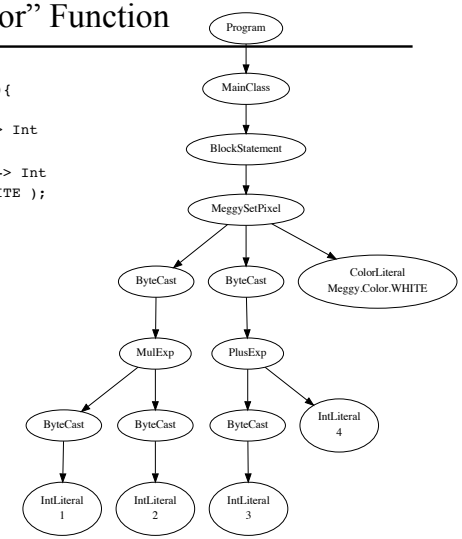
- Syntax-directed AST creation
- Function for creating the dot file for visualization
- Function for checking types (TODAY)
- Function for generating code (TODAY)

Later assignments

- Function for building a symbol table
- Function for allocating memory for variables
- Function for doing register allocation

Type Checking with a "Visitor" Function

```
class Byte {
  public static void main(String[] whatever){
    Meggy.setPixel(
      // Byte multiplication: Byte x Byte -> Int
      (byte)( (byte)1*(byte)2 ),
      // Mixed type expression: Byte x Int -> Int
      (byte)( (byte)3 + 4 ), Meggy.Color.WHITE );
  }
}
```



Type Checking

Java allows mixing numeric types. For Meggy-Java this means that many operators allow mixing byte and int

```
byte a = 0, b = 1;
byte b2 = (byte)-b;
byte wrong1 = -b;
byte sum = (byte)(a + 5);
byte wrong2 = a+5;
int ok = - - -b;
// but check out
int wrong3 = --b;
int j, i = 2;
if (i==b) j = i+b; else j=(byte)((byte)i+b);
```


Type Checking: valid operand types, result types

Signature of an operation (or function)

$inType_1$ x $inType_2$ … x $inType_n$ → outType

How to determine valid inTypes and resulting outType?

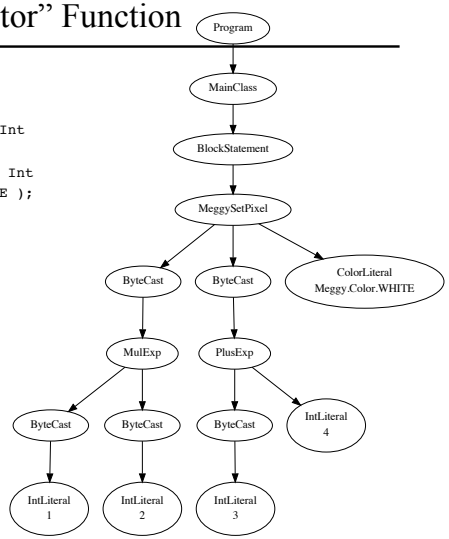
Use given reference compiler MJ.jar.

Create example test programs.

Let's look at an example …

Code Generation with a "Visitor" Function

```
class Byte {
  public static void main(String[] whatever){
    Meggy.setPixel(
      // Byte multiplication: Byte x Byte -> Int
      (byte)( (byte)1*(byte)2 ),
      // Mixed type expression: Byte x Int -> Int
      (byte)( (byte)3 + 4 ), Meggy.Color.WHITE );
  }
}
```



If Statement code generation

When the visitor encounters ifStmt, simple pre or post order code generation does not suffice. WHY?

We need more complex control:

if

**/ | **

B S1 S2

We need to control the order that code is generated for its children, using branches, jumps and labels.

First, code needs to be generated for the condition (the result of the condition evaluation has been pushed on the RTS) followed by branching instructions, the then block, control to jump over else block, then the else block, and then the end label.

Branches and jumps

An AVR detail: as you know from PA1, conditional branches can only go so far in the code, and code generated, e.g for then or else block is not bounded and thus can exceed that limit. Therefore we have to use jmp sometimes.

Notice: breq is replaced with with a brne followed by a jmp to handle this

```
cp     r24, r25
#WANT breq MJ_L6
brne   MJ_L7
jmp    MJ_L6
MJ_L7:
... inbounded stretch of code …
MJ_L6:
```


Not: there is no not in AVR, but there is xor

truth table for not and xor

x	y	!x	x xor y
0	0	1	0
0	1	1	1
1	0	0	1
1	1	0	0

We can implement NOT x with x XOR 1 :

outNotExp

```
pop     r24
ldi     r22, 1
eor     r24,r22
push    r24
```

While statement

while
**/ **
B S

What is the wiring logic?

```
SLbl:
      eval B on stack
      if false jump to endLbL
      gen Code for S
      jump to Slbl
   endLbL:
```


Short circuited (wired) AND, equals

Similar to the If Statement and While Statement, code generation will need to be implemented in the visitAndExp()

&&
**/ **
B1 B2

can be implemented as: if (B1) return B2 else return false

equalExp, the equality operator ==

Just like in plus and minus, we need to take the mixed type semantics of Java into account, by promoting a byte (1 register) to an int (register pair), making sure the int value correctly preserves the sign (remember the meggy AVR quiz).