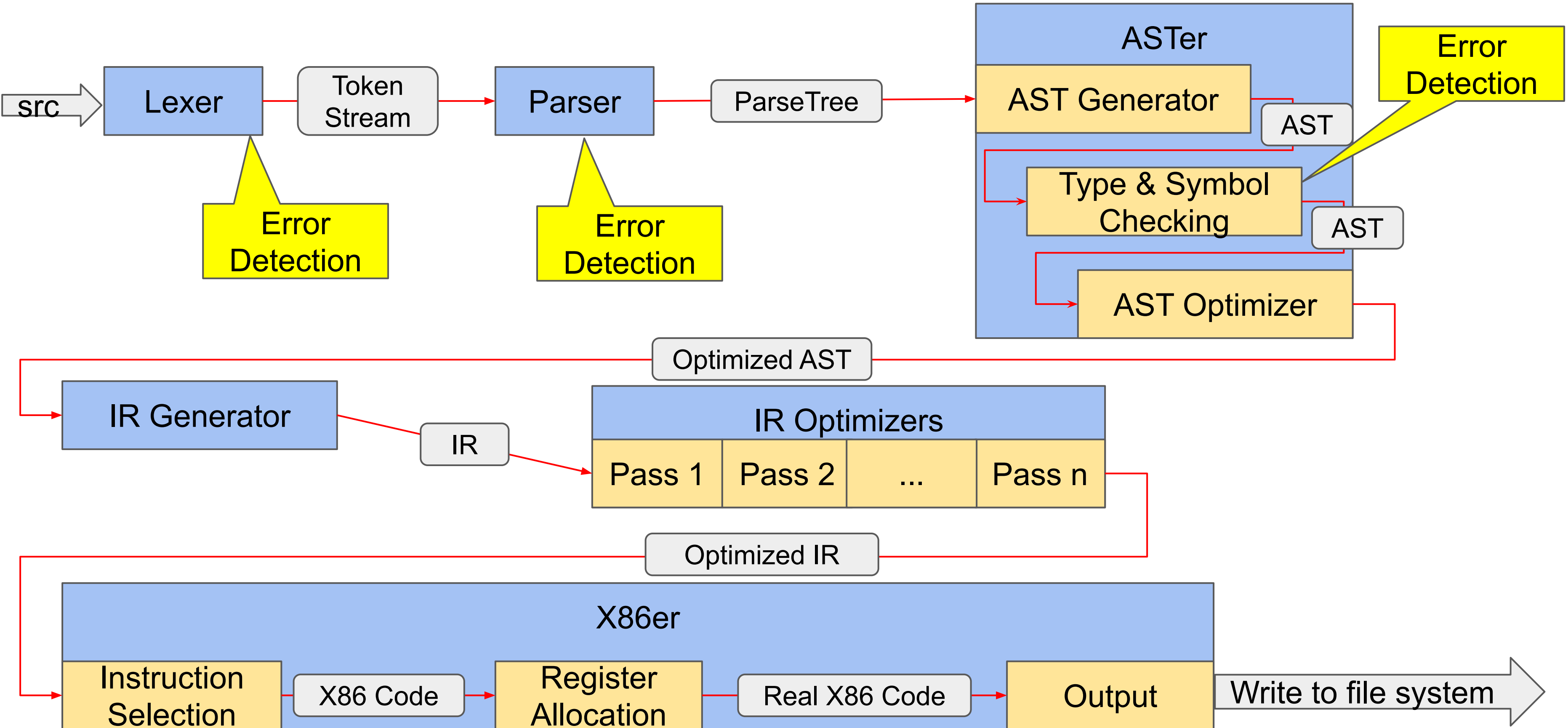
MiniJava Compiler

Michael & Jason

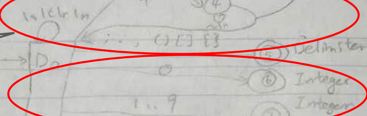
Workflow



Lexer

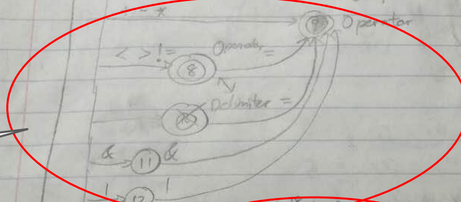
- Minimized DFA with 89 states
- Output a stream of tokens
- Each token has a type and a field
  - <RESERVED_WORD, class>
  - <INTEGER, 10>
  - <IDENTIFIER, myid>
  - <DELIMITER, ;>
  - <OPERATOR, +>
- Skips invalid characters and output error message

Comments

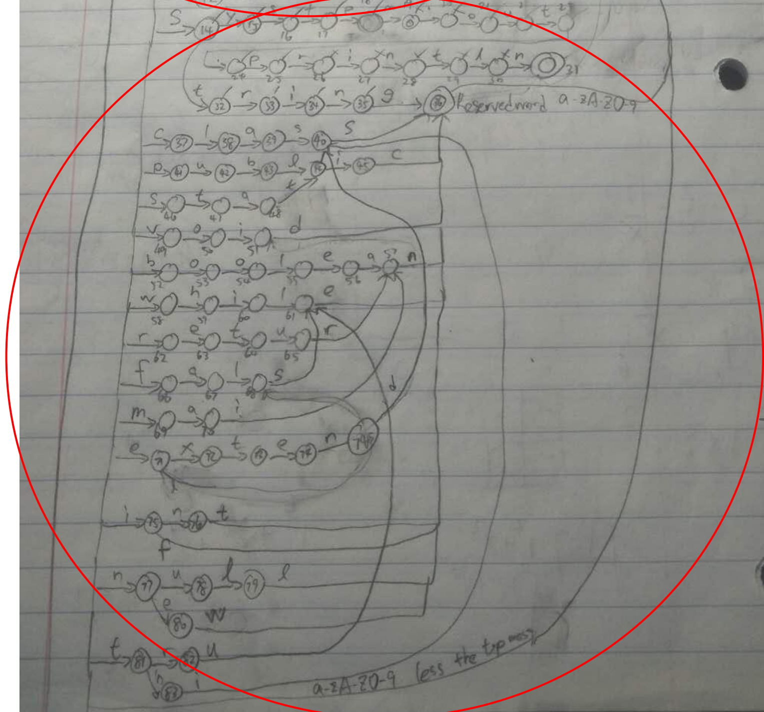


Integer

Operator &
Delimiter



ID



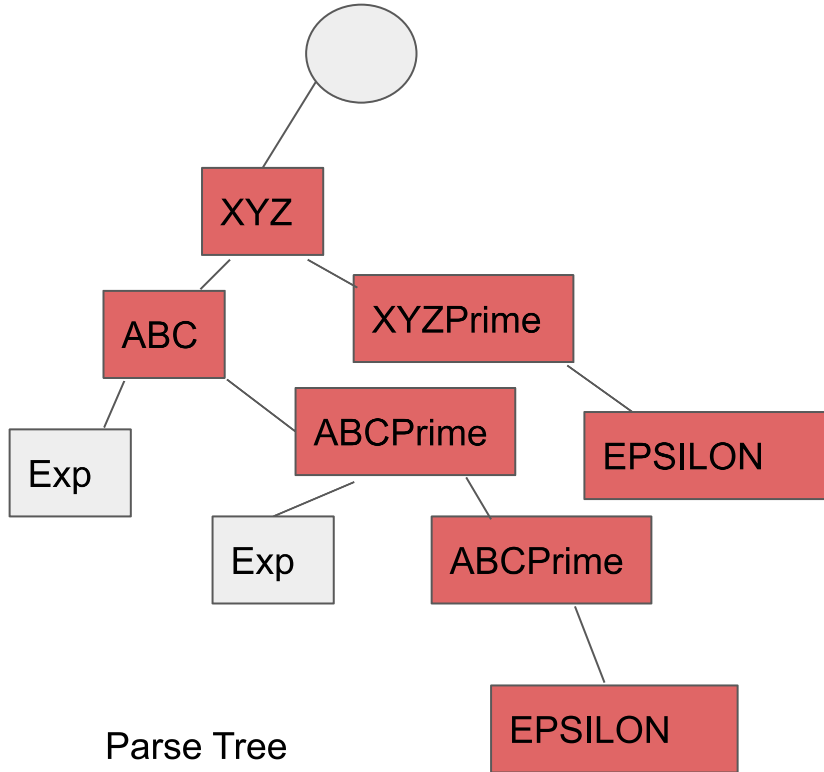
Reserved
Word

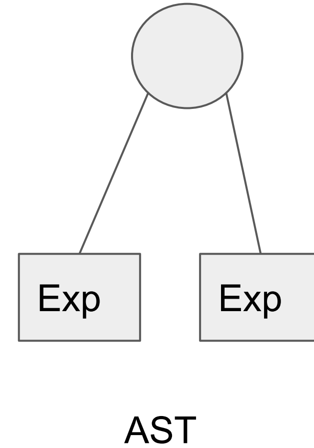
Parser

- Top-down Recursive Descent Parser
- Precedence encoded
- Forces local variable declaration at beginning of method
- Recovers from error by skipping tokens
- Output is a syntactically valid parse tree (hard to read)

```
mainClass: MainClass{
    className: BinarySearch,
    mainMethodParamName: a,
    statement: Statement{
        oneStatement: PrintStatement{
            expression: Expression{
                orTerm: OrTerm{
                    andTerm: AndTerm{
                        equalityTerm: EqualityTerm{
                            inequalityTerm: InequalityTerm{
                                term: Term{
                                    unaryTerm:
                                    ObjCreationUnaryTerm{
                                        objectCreation:
                                        UserDefinedTypeCreation{
                                            identifier: BS
                                        },
                                        invocation: Invocation{
                                            memberAccess:
                                            MethodInvokation{
                                                identifier: Start,
                                                t: Argument{
                                    }           ression:
                                },              ression{
                                termPrime: EPSILON  orTerm:
                            },
                            inequalityTermPrime: EPSILON
                        },
                        equalityTermPrime: EPSILON
                    },
                    andTermPrime: EPSILON
                },
                orTermPrime: EPSILON
            },
            expressionPrime: EPSILON
        }
    },
    statementPrime: EPSILON
}
},
```

Abstract Syntax Tree (AST)





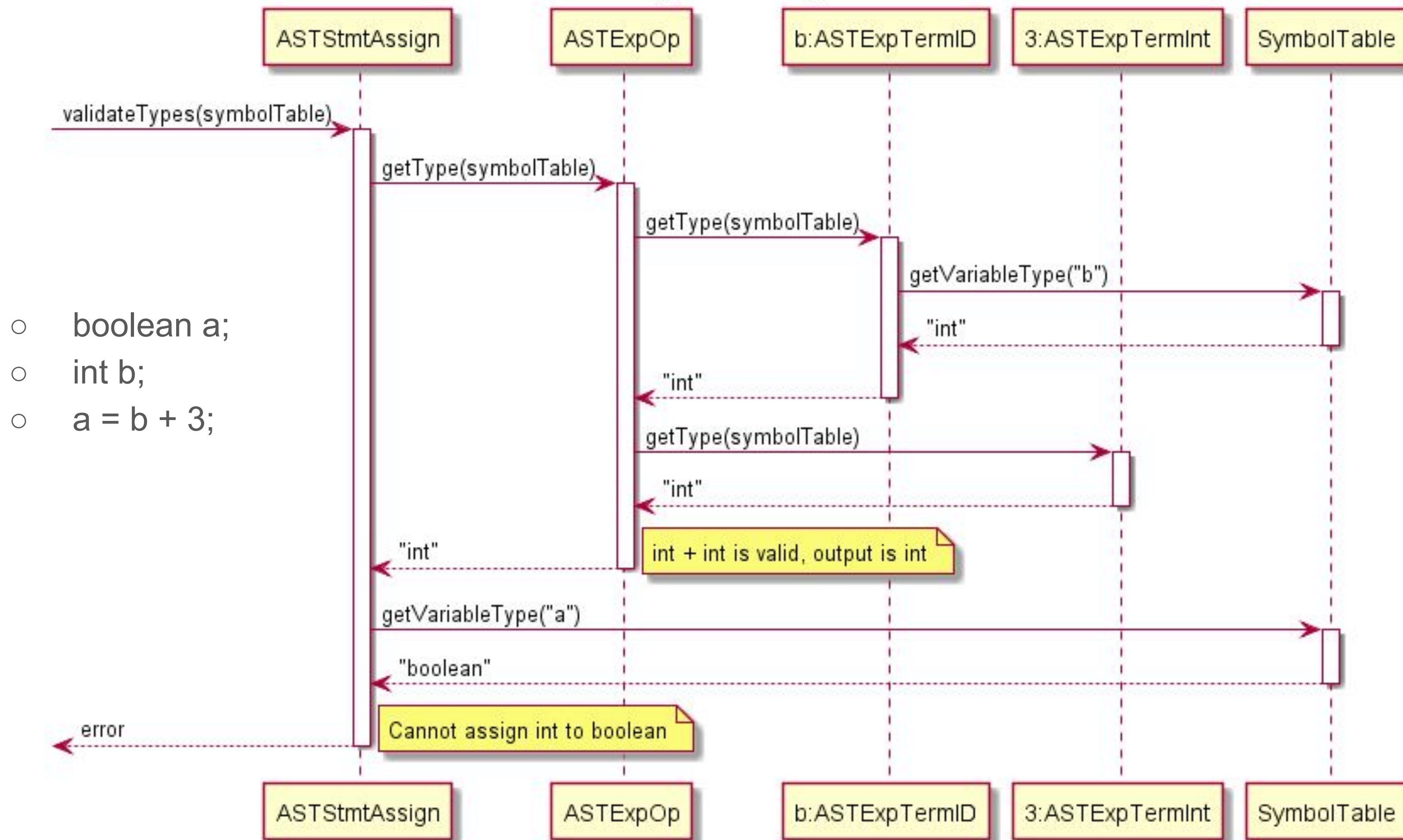
AST continued

3196 lines vs. 176 lines

```
MainClass:[name = BinarySearch, args = a]
    main():{
        ASTStmtPrint:[Invoke[obj = new BS, method = Start, args = [20]]
end main class
Class:[name = BS, super = null]
    fields:
        [type = int[], name = number]
        [type = int, name = size]
    methods:
        Method[returnType = int, name = Start]
            Args: [type = int, name = sz]
            Locals: [type = int, name = aux01][type = int, name = aux02]
            Statements:
                ASTStmtAssign:[ID = aux01, exp = Invoke[obj = <ID this>, method = Init, args = [<ID sz>]]
                ASTStmtAssign:[ID = aux02, exp = Invoke[obj = <ID this>, method = Print, args = []]
                ASTStmtIf:[cond = Invoke[obj = <ID this>, method = Search, args = [8]]
                    true:
                        ASTStmtPrint:[1]
                    false:
                        ASTStmtPrint:[0]
                end if
```

Type & Symbol Checking with AST

- Input is an AST of a syntactically correct program
- Recursively checks type
- Output is an AST of a semantically correct program
- Example:
  - boolean a;
  - int b;
  - a = b + 3;



Type Checking
ASTStmtAssign
ASTExpOp
b:ASTExpTermID
3:ASTExpTermInt
SymbolTable
validateTypes(symbolTable)
getType(symbolTable)
getType(symbolTable)
getVariableType("b")
"int"
"int"
getType(symbolTable)
"int"
"int"
int + int is valid, output is int
getVariableType("a")
"boolean"
error
Cannot assign int to boolean
boolean a;
int b;
a = b + 3;

AST Optimization

- Pre-computation
  - `int a = 3+5; -> int a = 8;`
- Algebraic/Logic Simplification
  - `Identity: a + 0 -> a; a * 1 -> a, etc`
  - `Shift: a * 2 -> a<<1; a / 4 -> a>>2, etc`
  - `Logic: true || a -> true; a || true -> true (but still evaluates a)`
  - `a || false -> a, etc`
- Dead Conditional
  - `if (true) {ABC} else {XYZ} -> ABC`
  - `While (false) {ABC} -> NOTHING`

Short circuit is also implemented:

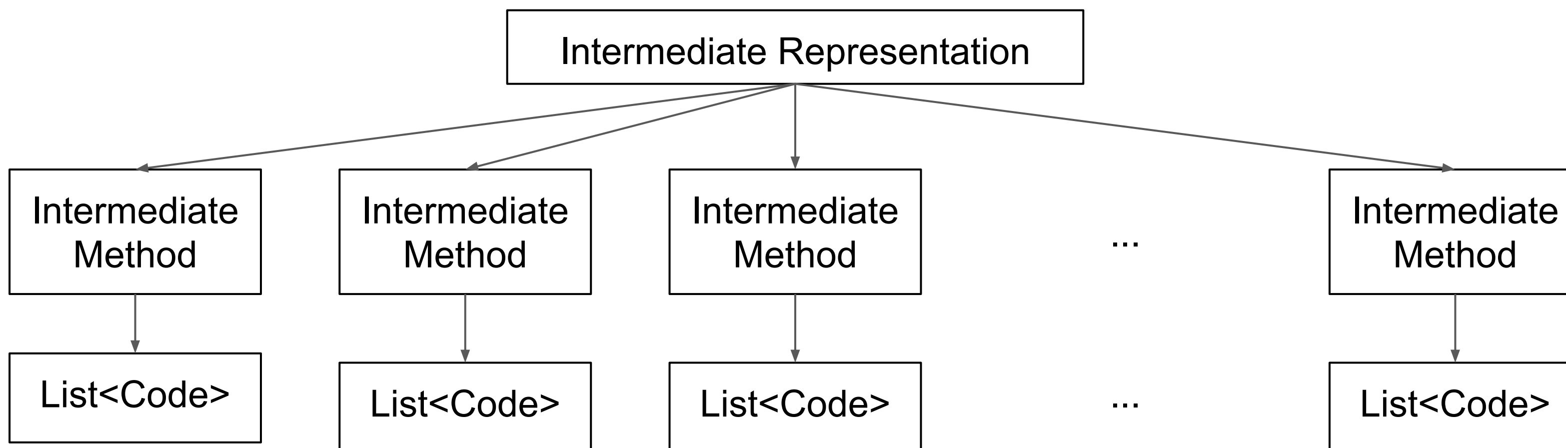
a && b => if (!a) false else b

```
b1 = b1&&b2||b3&&b4;
```

```
    STORE     <Var b1> <Var @15>
    JE        <Label @16> <Var @15> <Int 0>
    AND       <Var @15> <Var b2>
<Label @16>:
    STORE     <Var @15> <Var @13>
    JE        <Label @14> <Var @13> <Int 1>
    STORE     <Var b3> <Var @17>
    JE        <Label @18> <Var @17> <Int 0>
    AND       <Var @17> <Var b4>
<Label @18>:
    OR        <Var @13> <Var @17>
<Label @14>:
    STORE     <Var @13> <Var b1>
```

Intermediate Representation (IR)

The program is represented using a list of Intermediate Methods.
Each method has a record of its arguments, local variables, etc.



Intermediate code generated is assembly-like and follows a MIPS-like syntax
Example: SUB <Var @51> <Var num> <Int 1> // @51 = num - 1;

Temporary variables are generated in this step (prefix '@' followed by a serial number)
Fields are replaced with load/store at an offset

IR Optimization

Optimization	Description	Example
Operation + Store	Optimizes operations that output a variable that is immediately stored to somewhere else.	ADD <Var @1> <Var a> <Int 3> STORE <Var b> ⇒ ADD <Var b> <Var a> <Int 3>
Set&Jmp	Combines Setcc and Je with Jcc	SETLE <Var @1> <Var a> <Var b> JE <LABEL> <Var @1> <Int 0> ⇒ JGE <LABEL> <Var a> <Var b>
Unused Label	Labels that are not referred to by jumps.	
Duplicate Labels	Labels that are duplicated	<LABEL l1>: <LABEL l2>: ⇒ Removed JMP <LABEL l2> ⇒ JMP <LABEL l1>
Jump Propagation	Jumping to a jump or jumping to next line	JMP <LABEL 1>: ⇒ JMP <LABEL 2> … <LABEL 1>: JMP <LABEL 2>

X86 Instruction Selection

- Must deal with gory details regarding valid operand types (Var vs. Int)
  - Mov instructions are inserted as needed
- Arithmetic/logic instructions map easily to X86 instructions
  - May need to convert from 3-address to 2-address
- Procedure call is more involved:
  - Backing up registers
  - Pushing arguments
  - Preparing call information for call handler (later)
  - Call the call handler
  - Get return value
  - Throw away arguments
  - Restoring registers

```
Fac$ComputeFac:
    MAKE_FRAME
    LOAD_A     <Var this> <Var num>
    JGE        <Label @3> <Var num> <Int 1>
    STORE      <Int 1> <Var num_aux>
    JMP        <Label @4>
<Label @3>:
```

```
Fac$ComputeFac:
    <Hint>  Allocate Stack Space Here</Hint>
    movq    24(%rbp), this
    movq    32(%rbp), num
    cmpq    $1, num
    jge .L_3
    movq    $1, num_aux
    jmp .L_4
.L_3:
```

X86 Register Allocation

Extremely simple algorithm:

1. Use AX only for return values and compiler temporary.
2. Pick 8 arbitrary variables and use 8 registers (r8 - r15) for them.
3. Put the rest on the stack.

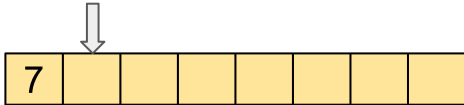
```
movq    %rax, @1
movq    $24, 0(@1)
leaq    BS$Start(%rip), %rax
movq    %rax, 24(@1)
leaq    BS$Search(%rip), %rax
movq    %rax, 32(@1)
leaq    BS$Div(%rip), %rax
movq    %rax, 40(@1)
leaq    BS$Compare(%rip), %rax
movq    %rax, 48(@1)
leaq    BS$Print(%rip), %rax
movq    %rax, 56(@1)
leaq    BS$Init(%rip), %rax
movq    %rax, 64(@1)
```

```
movq    $24, 0(%r9)
leaq    BS$Start(%rip), %rax
movq    %rax, 24(%r9)
leaq    BS$Search(%rip), %rax
movq    %rax, 32(%r9)
leaq    BS$Div(%rip), %rax
movq    %rax, 40(%r9)
leaq    BS$Compare(%rip), %rax
movq    %rax, 48(%r9)
leaq    BS$Print(%rip), %rax
movq    %rax, 56(%r9)
leaq    BS$Init(%rip), %rax
movq    %rax, 64(%r9)
```

Array Representation

Integers, booleans and pointers are all 8 bytes (for simplicity)

Arrays are stored as pointers. A[0] is the first element. A[-1] is length of the array

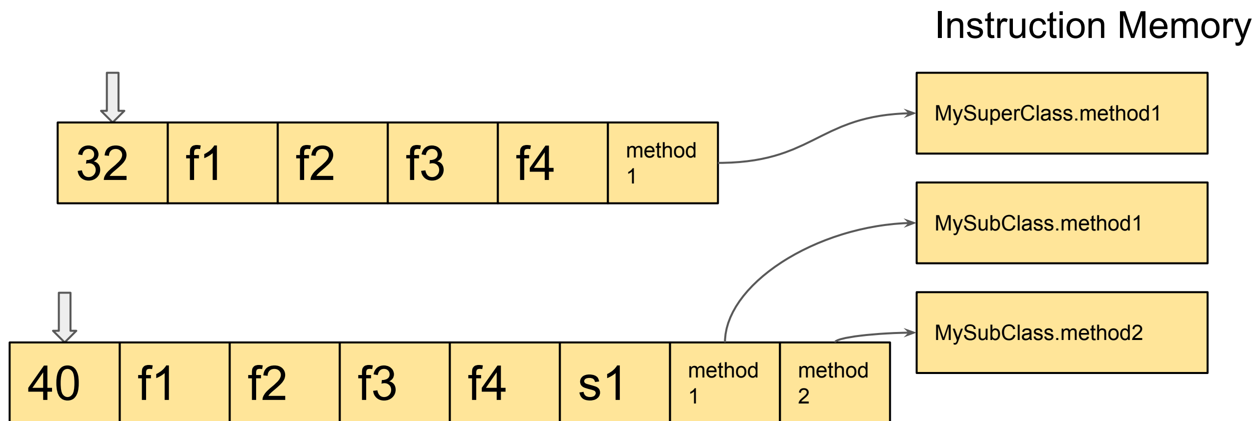


Object Representation

Recall that integers, booleans and pointers are all 8 bytes (for simplicity)

Objects are stored in contiguous memory as fields and method pointers. Access to fields and methods are determined at compile time

```
class MySuperClass{
    int f1;
    int f2;
    int[] f3;
    MySuperClass f4;
    public int method1(){
        ...
    }
}
class MySubClass extends MySuperClass{
    boolean s1;
    public int method2(){
        ...
    }
    //Override
    public int method1(){
        ...
    }
}
```



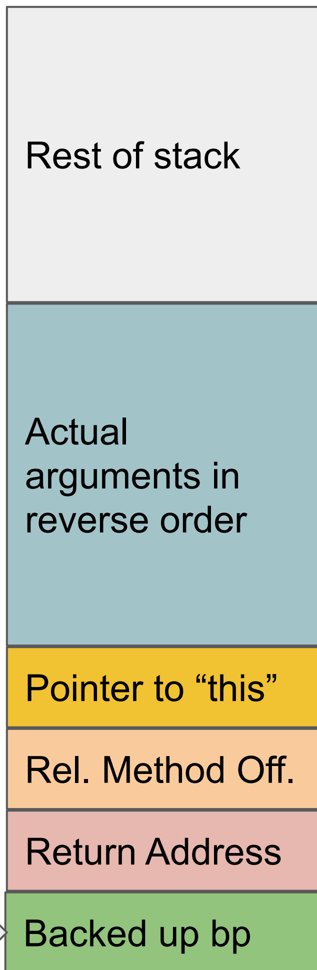
Note that fields have consistent offsets from the beginning, while methods have consistent offsets from the first method, which can be calculated from the field sizes

Call Handler

- Method calls cannot simply use CALL instructions. They must support dynamic binding
- The call handler function

```
    .text
    .global chandle
    .type chandle, @function
chandle:
    pushq   %rbp
    movq    %rsp, %rbp
    pushq   %rbx                    ;Backup rbx
    movq    24(%rbp), %rax          ;rax = &this
    movq    0(%rax), %rbx           ;rbx = absolute offset of first method
    addq    16(%rbp), %rbx          ;rbx = absolute offset of the method called
    addq    %rbx, %rax              ;rax = rbx + &this = absolute address of the method pointer
    popq    %rbx
    jmp *(%rax)                     ;Jump to method
    .size chandle, .-chandle
    .text
```

bp



Garbage Dumping: Be nice to the OS

- Record allocated regions and free them all at once when program exits
- Use a global linked stack
- Pros: Easy to implement in assembly
- Cons: Nothing will be freed if the program doesn’t finish running

Running valgrind on TreeVisitor:

```
==3616==
==3616== HEAP SUMMARY:
==3616==     in use at exit: 0 bytes in 0 blocks
==3616==   total heap usage: 21 allocs, 21 frees, 3,024 bytes alloca
ted
==3616==
==3616== All heap blocks were freed -- no leaks are possible
```

Additional Features to Implement (if we had time*)

- Target language optimization
  - Peephole optimization
- Improve instruction selection
- Improve register allocation
- Improve error detection / handling

* **Might** be able to do some of these by Friday