07-0: **Abstract Assembly Trees**

- Once we have analyzed the AST, we can start to produce code
- We will *not* produce actual assembly directly – we will go through (yet another) internal representation – Abstract Assembly Trees
    - Translating from AST to assembly is difficult – much easier to translate from AST to AAT, and (relatively) easy to translate from AAT to assembly.
    - Optimizations will be easier to implement using AATs.
    - Writing a compiler for several different targets (i.e., x86, MIPS) is much easier when we go through AATs

07-1: **Implementing Variables**

- In simpleJava, all local variables (and parameters to functions) are stored on the stack.
    - (Modern compilers use registers to store local variables wherever possible, and only resort to using the stack when absolutely necessary – we will simplify matters by always using the stack)
- Class variables and arrays are stored on the heap (but the *pointers* to the heap are stored on the stack)

07-2: **Activation Records**

- Each function has a segment of the stack which stores the data necessary for the implementation of the function
    - Mostly the local variables of the function, but some other data (such as saved register values) as well.
- The segment of the stack that holds the data for a function is the "Activation Record" or "Stack Frame" of the function

07-3: **Stack Implementation**

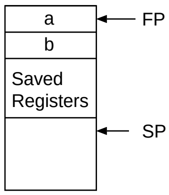
- Stack is implemented with two registers:
    - Frame Pointer (FP) points to the beginning of the current stack. frame
    - Stack Pointer (SP) points to the next free address on the stack.
- Stacks grow from large addresses to small addresses.

07-4: **Stack Frames**

- The stack frame for a function `foo()` contains:
    - Local variables in `foo`
    - Saved registers & other system information
    - Parameters of functions *called by* `foo`

07-5: **Stack Frames**

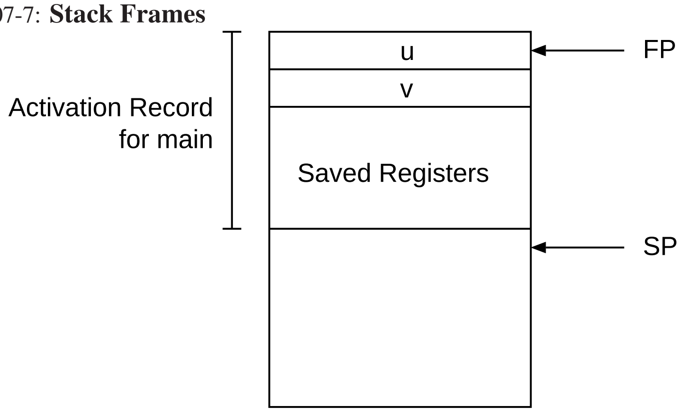
```
int foo() {
   int a;
   int b;

   /* body of foo */
}
```



07-6: **Stack Frames**

```
void foo(int a, int b);
void bar(int c, int d);

void main() {                    int foo(int c, int d) {
   int u;                           int x;
   int v;                           int y;

   /* Label A */                    /* Label B */
                                 }
   bar(1,2);
}

void bar(int a, int b) {
   int w;
   int x;

   foo(3,4);
}
```

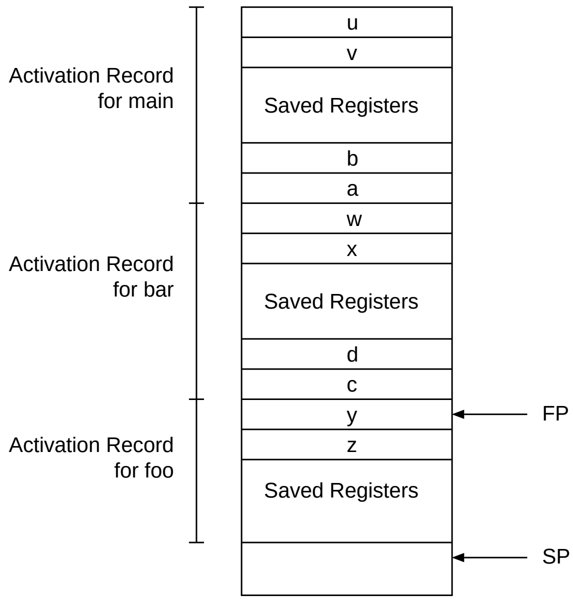
07-7: **Stack Frames**



07-8: **Stack Frames**

```
void foo(int a, int b);
void bar(int c, int d);

void main() {                    int foo(int c, int d) {
   int u;                           int x;
   int v;                           int y;

   /* Label A */                    /* Label B */
                                 }
   bar(1,2);
}

void bar(int a, int b) {
   int w;
   int x;

   foo(3,4);
}
```

07-9: **Stack Frames**



07-10: **Accessing Variables**

- Local variables can be accessed from the frame pointer
  - Subtract the offset of the variable from the frame pointer
  - (remember – stacks grow down!)
- Input parameters can also be accessed from the frame pointer
- Add the offset of the parameter to the frame pointer

07-11: **Setting up stack frames**

- Each function is responsible for setting up (and cleaning up) its own stack frame
- Parameters are in the activation record of the calling function
  - Calling function places parameters on the stack
  - Calling function cleans up parameters after the call (by incrementing the Stack Pointer)

07-12: **Abstract Assembly**

- There are two kinds of Assembly Trees:
  - Expression Trees, which represent values
  - Statement Trees, which represent actions
- Just like Abstract Syntax Trees

07-13: **Expression Trees**

- Constant Expressions
  - Stores the value of the constant.
  - Only integer constants

- (booleans are represented as integers, just as in C)

07-14: **Expression Trees**

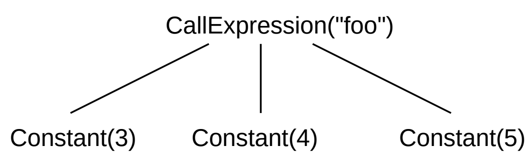
- Register Expressions
  - Contains a description of the register
    - Stack Pointer (SP)
    - Frame Pointer (FP)
    - Result Register (for return value of functions)
    - Return Register (for return address of function calls)

07-15: **Expression Trees**

- Operator Expressions
  - Contains the operator, left subtree, and right subtree
    - +, -, *, /, $<$, $\leq$, $>$, $\geq$, &&, $||$, !

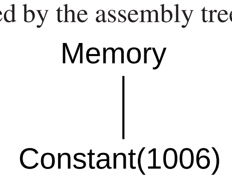
07-16: **Expression Trees**

- Call Expression
  - Contains the assembly language for the start of the function, and an expression for each actual parameter
  - `foo(3,4,5)`



07-17: **Expression Trees**

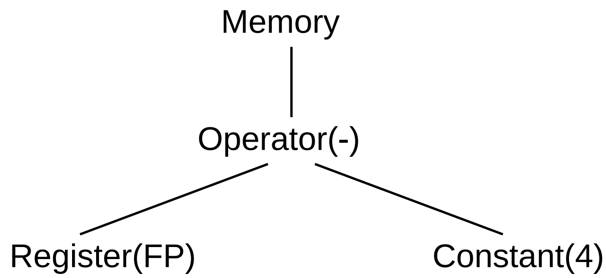
- Memory Expression
  - Represents a memory dereference. Contains the memory location to examine.
  - Memory location 1006 is represented by the assembly tree:



07-18: **Expression Trees**

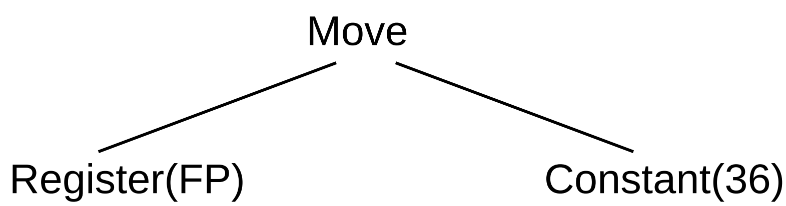
- Memory Expression
  - Represents a memory dereference. Contains the memory location to examine.

- Local variable with an offset of 4 off the FP is



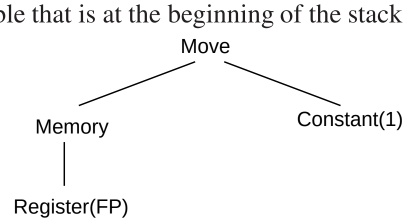
07-19: **Statement Trees**

- Move Statements
  - Move statements are used to move data into either a memory location or a register
  - Left subtree of a move statement must be a register or memory expression
  - Right subtree of a move statement is any expression
  - To store the value 36 in the Frame Pointer:



07-20: **Statement Trees**

- Move Statements
  - Move statements are used to move data into either a memory location or a register
  - Left subtree of a move statement must be a register or memory expression
  - Right subtree of a move statement is any expression
  - To store the value 1 a variable that is at the beginning of the stack frame:



07-21: **Statement Trees**

- Label Statements
  - A Label statement represents an assembly language label.
  - Used by jumps, conditional jumps, and function/procedure calls

07-22: **Statement Trees**

- Jump Statements
  - Unconditional jump
  - Contains an assembly language label
  - Control is immediately transferred to the new location

07-23: **Statement Trees**

- ConditionalJump Statements
  - Contains an assembly language label, and an expression subtree
  - If the expression is true (non-zero), then control is transferred to the assembly language label
  - If the expression is false (zero), the conditional jump is a no-op

07-24: **Statement Trees**

- Sequential Statements
  - Contain two subtrees – a left subtree and a right subtree
  - First the left subtree is executed, then the right subtree is executed.

07-25: **Statement Trees**

- Call Statements
  - Just like Call Expressions, except they return no value
  - Contain an assembly language label, and a list of expressions that represent actual parameters

07-26: **Statement Trees**

- Empty Statement
  - No-op
  - Empty statements make creating assembly for statements that do nothing easier
    - What simpleJava statements produce no assembly?

07-27: **Statement Trees**

- Empty Statement
  - No-op
  - Empty statements make creating assembly for statements that do nothing easier
    - What simpleJava statements produce no assembly?
    - Variable declaration statements (that have no initialization)
  - In code generation phase, these statements will be dropped

07-28: **Statement Trees**

- Return Statements
  - Return flow of control to the calling procedure
  - Do *not* return a value, *only* changes the flow of control

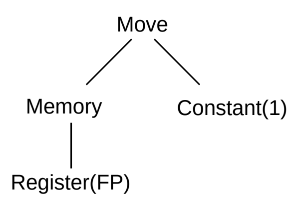
- A simpleJava return statement will be implemented with extra Abstract Assembly to handle setting the return value of the function

07-29: **Abstract Assembly Examples**

```
void foo(int a, int b) {
   int x;
   int y;
   boolean z;

   x = 1;          <--- This statement
   y = a * b;
   y++;
   bar(y, x + 1, a);
   x = function(y+1, 3);
   if (x > 2)
      z = true;
   else
      z = false;
}
```

07-30: **Abstract Assembly Examples**

```
x = 1;
```

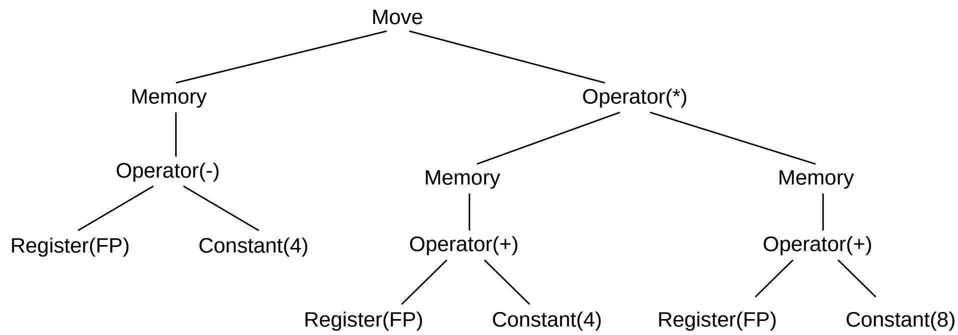


07-31: **Abstract Assembly Examples**

```
void foo(int a, int b) {
   int x;
   int y;
   boolean z;

   x = 1;
   y = a * b;    <--- This statement
   y++;
   bar(y, x + 1, a);
   x = function(y+1, 3);
   if (x > 2)
      z = true;
   else
      z = false;
}
```

07-32: **Abstract Assembly Examples**

```
y = a * b;
```

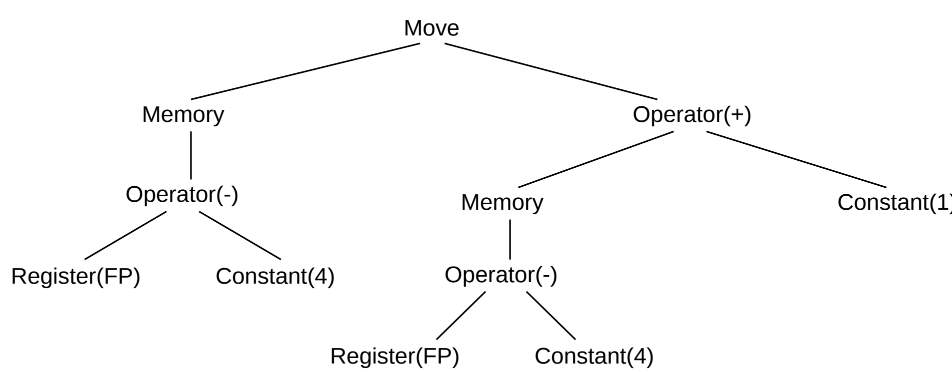


07-33: **Abstract Assembly Examples**

```
void foo(int a, int b) {
   int x;
   int y;
   boolean z;

   x = 1;
   y = a * b;
   y++;             <--- This statement
   bar(y, x + 1, a);
   x = function(y+1, 3);
   if (x > 2)
      z = true;
   else
      z = false;
}
```

07-34: **Abstract Assembly Examples**

```
y++;
```



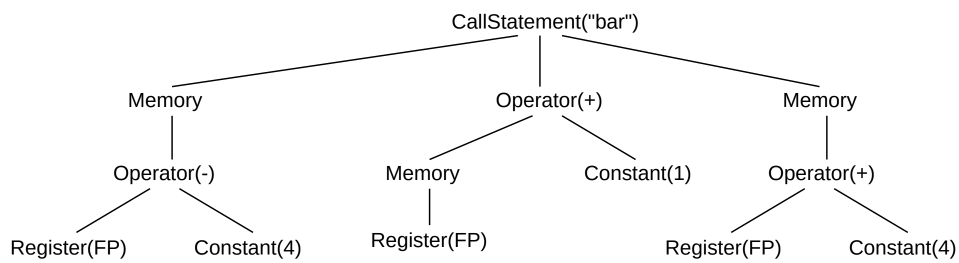
07-35: **Abstract Assembly Examples**

```
void foo(int a, int b) {
   int x;
   int y;
   boolean z;
```

```
   x = 1;
   y = a * b;
   y++;
   bar(y, x + 1, a);      <--- This statement
   x = function(y+1, 3);
   if (x > 2)
      z = true;
   else
      z = false;
}
```

07-36: **Abstract Assembly Examples**

```
bar(y, x + 1, a);
```

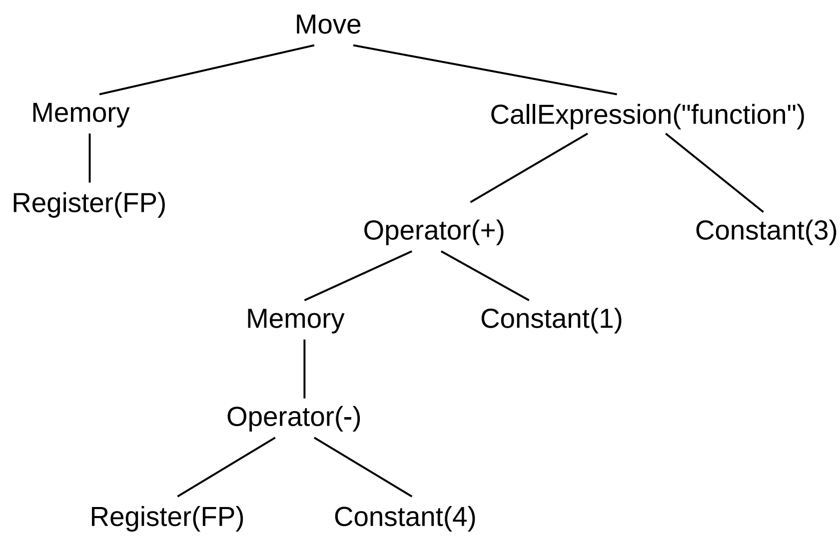


07-37: **Abstract Assembly Examples**

```
void foo(int a, int b) {
   int x;
   int y;
   boolean z;

   x = 1;
   y = a * b;
   y++;
   bar(y, x + 1, a);
   x = function(y+1, 3);  <--- This statement
   if (x > 2)
      z = true;
   else
      z = false;
}
```

07-38: **Abstract Assembly Examples**

```
x = function(y+1, 3);
```

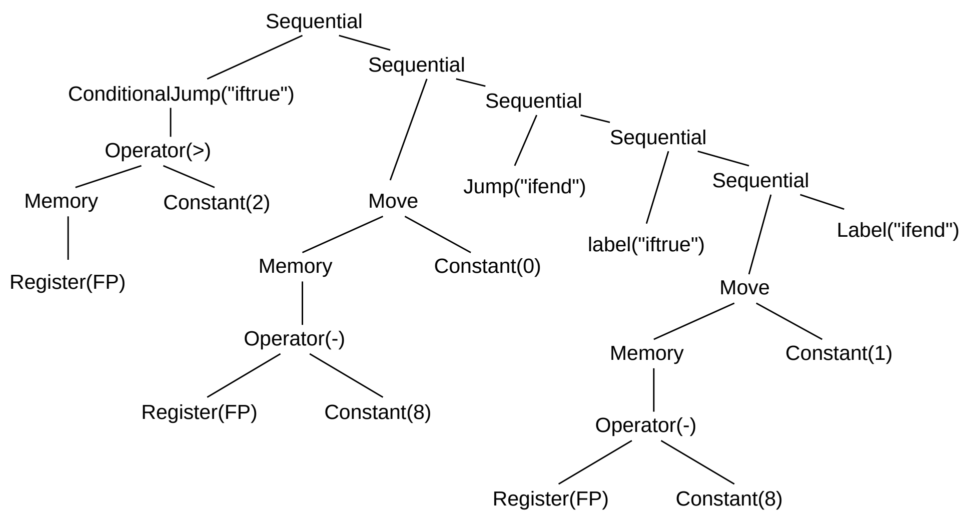


07-39: **Abstract Assembly Examples**

```
void foo(int a, int b) {
   int x;
   int y;
   boolean z;

   x = 1;
   y = a * b;
   y++;
   bar(y, x + 1, a);
   x = function(y+1, 3);
   if (x > 2)            -|
      z = true;           | If statement
   else                   |
      z = false;         -|
}
```

07-40: **Abstract Assembly Examples**

```
if (x > 2) z = true; else z = false;
```

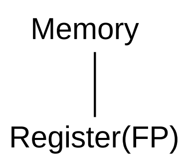


07-41: **Creating Abstract Assembly**

- Base Variables (x, y, etc.)
  - Stored on the stack
  - Accessed through the Frame Pointer

```
void foo() {
    int x;
    int y;
    /* Body of foo */
}
```

  - Assembly tree for x:

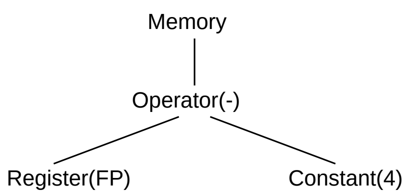


07-42: **Creating Abstract Assembly**

- Base Variables (x, y, etc.)
  - Stored on the stack
  - Accessed through the Frame Pointer

```
void foo() {
    int x;
    int y;
    /* Body of foo */
}
```

  - Assembly tree for y:

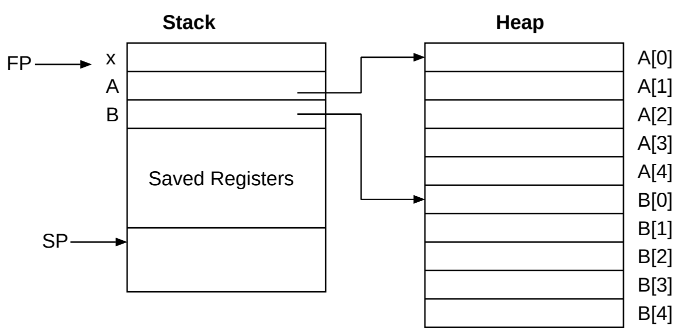


07-43: **Creating Abstract Assembly**

- Array Variables (A[3], B[4][5], etc)
  - Contents of array are stored on the heap
  - Pointer to the base of the array is stored on the stack

07-44: **Array Variables**

```
void arrayallocation() {
    int x;
    int A[] = new int[5];
    int B[] = new int[5];
    /* body of function */
}
```



07-45: **Array Variables**

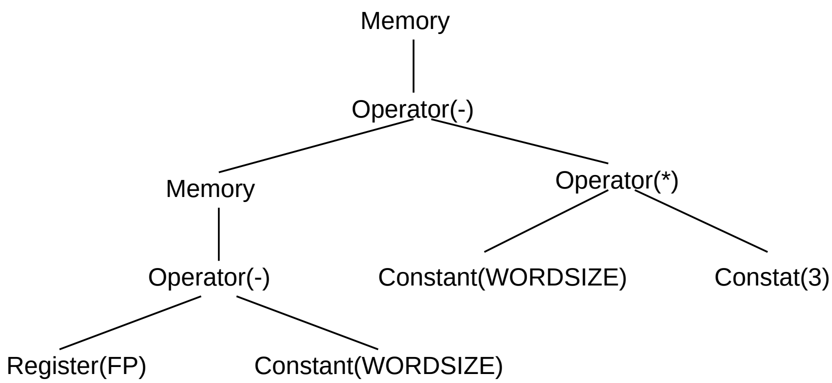
- How do we represent A[3] in abstract assembly?
  - Use the offset of `A` to get at the beginning of the array
  - Subtract 3 * (element size) from this pointer
    - In simpleJava, all variables take a single word. Complex variables – classes and arrays – are pointers, which also take a single word
    - Heap memory works like stacks – "grow" from large addresses to small addresses
  - Dereference this pointer, to get the correct memory location

07-46: **Array Variables**

- `A[3]`

07-47: **Array Variables**

- `A[3]`

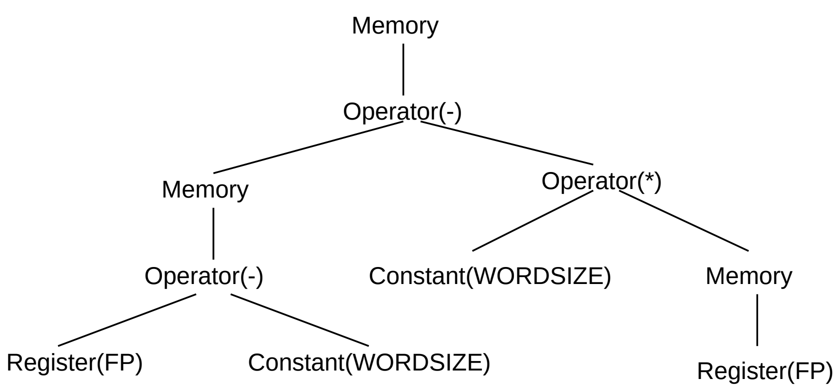


07-48: **Array Variables**

- `A[x]`

07-49: **Array Variables**

- `A[x]`

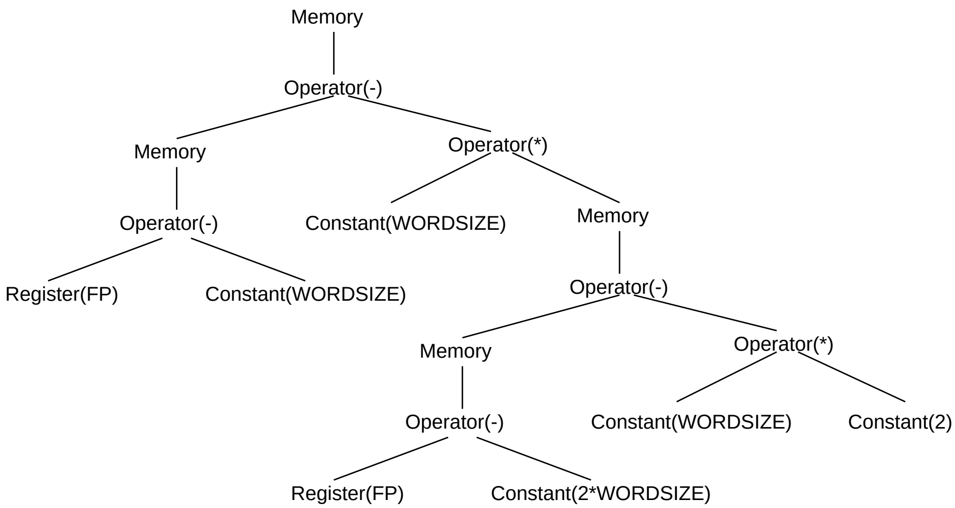


07-50: **Array Variables**

- `A[B[2]]`

07-51: **Array Variables**

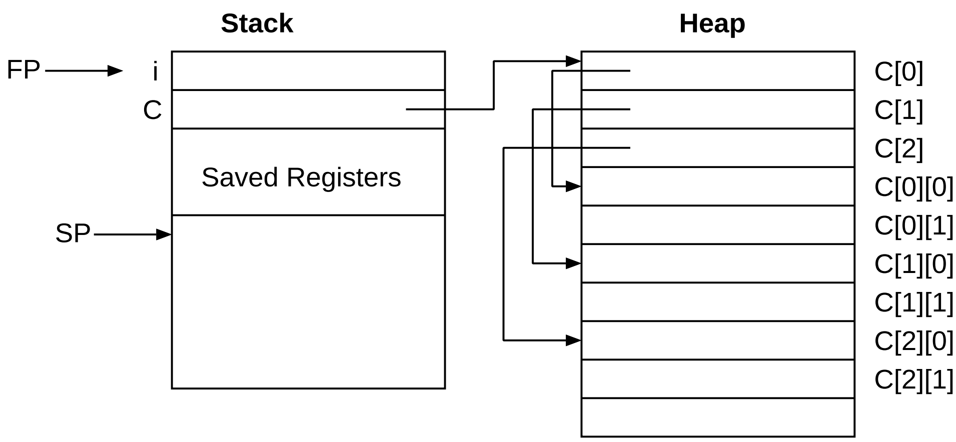
- `A[B[2]]`



07-52: **2D Arrays**

```
void twoDarray {
   int i;
   int C[][];

   C = new int[3][];
   for (i=0; i<3; i++)
      C[i] = new int[2];

   /* Body of function  */
}
```

07-53: **2D Arrays**

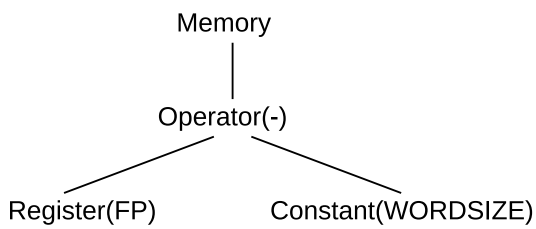


07-54: **2D Arrays**

- C

07-55: **2D Arrays**

- C

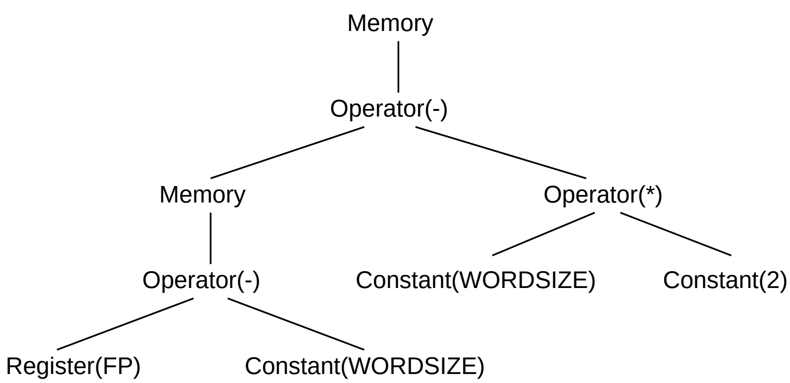


07-56: **2D Arrays**

- C[2]

07-57: **2D Arrays**

- C[2]

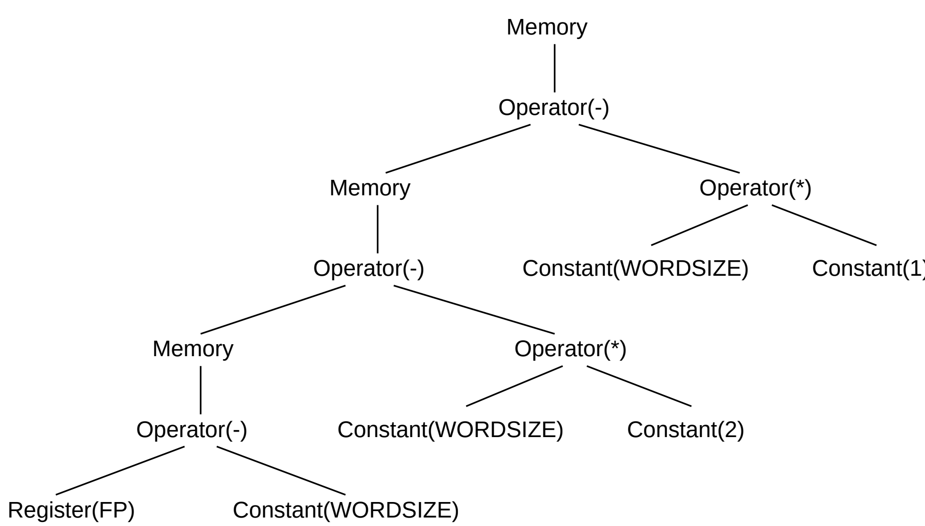


07-58: **2D Arrays**

- C[2][1]

07-59: **2D Arrays**

- C[2][1]



07-60: **Instance Variables**

- `x.y`, `z.w`
- Very similar to array variables
  - Array variables – offset needs to be calculated
  - Instance variables – offset known at compile time

07-61: **Instance Variables**

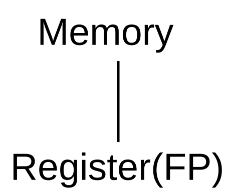
```
class simpleClass {
   int x;
   int y;
   int A[];
}

void main() {
   simpleClass s;
   s = new simpleClass();
   s.A = new int[3];

   /* Body of main */
}
```

07-62: **Instance Variables**

- Variable `s`

07-63: **Instance Variables**

- Variable `s`

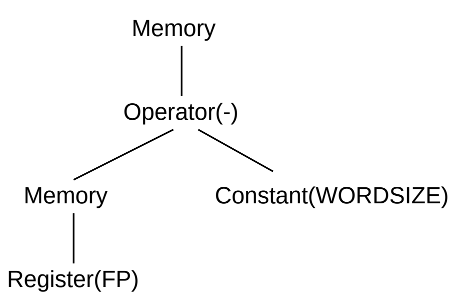


07-64: **Instance Variables**

- Variable `s.y`

07-65: **Instance Variables**

- Variable `s.y`

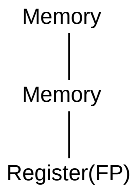


07-66: **Instance Variables**

- Variable `s.x`

07-67: **Instance Variables**

- Variable `s.x`

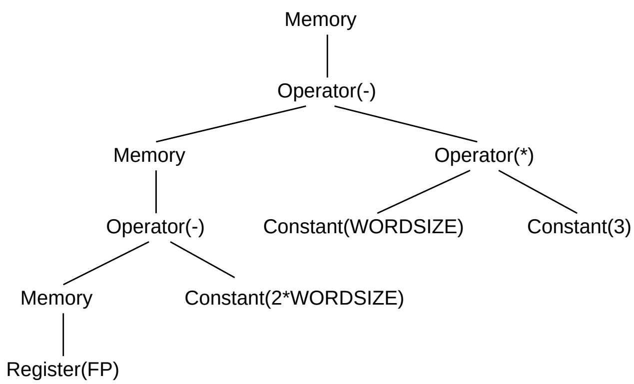


07-68: **Instance Variables**

- Variable `s.A[3]`

07-69: **Instance Variables**

- Variable `s.A[3]`

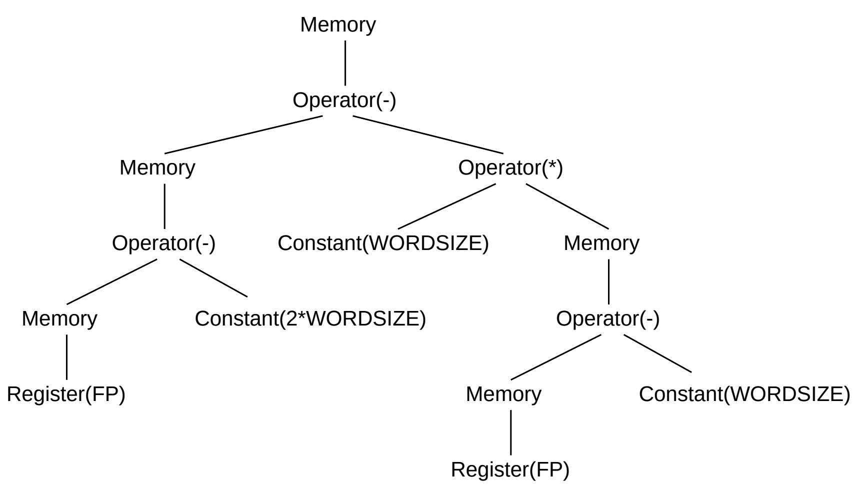


07-70: **Instance Variables**

- Variable `s.A[s.y]`

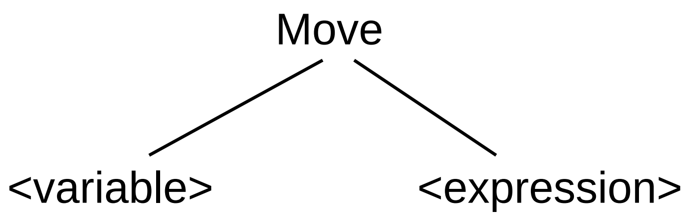
07-71: **Instance Variables**

- Variable `s.A[s.y]`



07-72: **Statements**

- Assignment Statements
    - ¡variable¿ = ¡expression¿



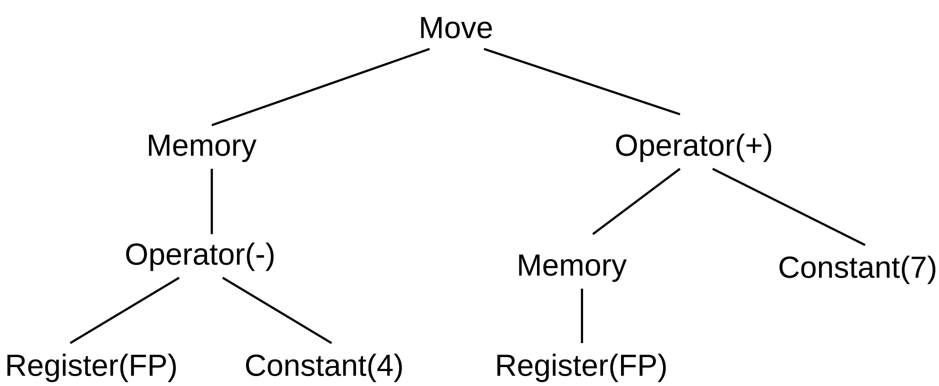
07-73: **Assignment Statements**

```
void main() {
   int x;
   int y;

   y = x + 7;
}
```

- Assembly tree for `y = x + 7` ?

07-74: **Assignment Statements**

```
y = x + 7;
```



07-75: **If Statements**

```
if (<test>) <statement1> else <statement2>
```

07-76: **If Statements**

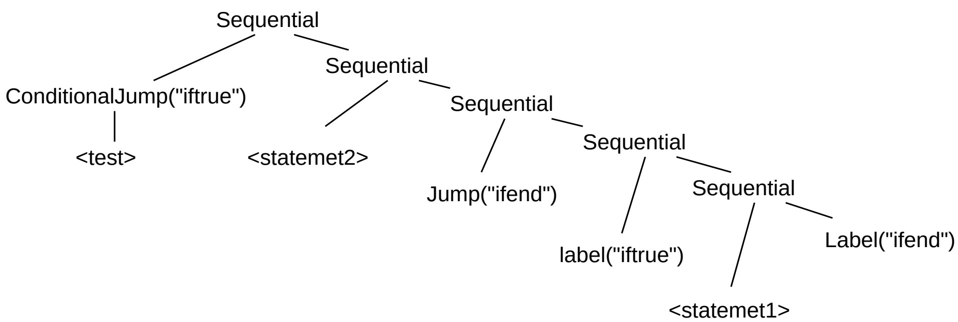
```
if (<test>) <statement1> else <statement2>
```

- Pseudo-Assembly:

```
       If (<test>) goto IFTRUE
       < code for statement2 >
        goto IFEND
IFTRUE:
       < code for statement1 >
IFEND:
```

07-77: **If Statements**

```
if (<test>) <statement1> else <statement2>
```



07-78: **While Statements**

```
while (<test>) <statement>
```

07-79: **While Statements**

```
while (<test>) <statement>
```

- Straightforward version:

```
WHILESTART:
       If (not <test>) goto WHILEEND
       < code for statement >
        goto WHILESTART
WHILEEND:
```

07-80: **While Statements**

```
while (<test>) <statement>
```

- More Efficient:

```
       goto WHILETEST
WHILESTART:
       < code for statement >
WHILETEST:
       If (<test>) goto WHILESTART
```

07-81: **For Statements**

```
for (<initialize>;<test>;<increment>)
    <statement>
```

07-82: **For Statements**

```
for (<initialize>;<test>;<increment>)
    <statement>
```

- Equivalent to:

```
<initialize>
while (<test>) {
    <statement>
    <increment>
}
```

07-83: **For Statements**

```
for (<initialize>;<test>;<increment>) <statement>
```

- Pseudo-Assembly:

```
   <initialize>
   goto FORTEST
FORSTART:
   <statement>
   <increment>
FORTEST:
   if (<test>) goto FORSTART
```

07-84: **Return Statements**

- To implement a return statement:
  - Copy the value of the return statement into the Result register
  - Clean up
    - Restore saved registers
    - Pop off current activation record
    - Return control to calling function

07-85: **Return Statements**

- To implement a return statement:
  - Copy the value of the return statement into the Result register
  - Clean up
    - Jump to end of function (where the cleanup code lives)

07-86: **Function Definitions**

- What Code is necessary to implement a function definition?
  - Assembly language label for the start of the function
  - Code to set up the activation record for the function
  - The actual code of the function itself
  - An assembly language label (to jump to on return statements)
  - Code to clean up the activation record for the function
  - A “Return” assembly language instruction

07-87: **Function Definitions**

- Setting up the activation record:
    - Store old values of Stack Pointer, Frame Pointer, and Return Address registers on the stack.
        - Use the Stack Pointer, not the Frame Pointer, to access the correct location in the activation record (why?)
        - Remember to leave space for the local variables in the function!
    - Point the Frame Pointer to the beginning of the current activation record
    - Point the Stack Pointer to the end of the current activation record
        - Leave space for local vars & saved registers!

07-88: **Function Definitions**

- Cleaning up the activation record:
    - Restore old values of the Stack Pointer, Frame Pointer, and Return Address registers
        - Be careful to do this in the correct order!
        - If you use the Frame Pointer to get at items on the stack, it should be restored *last*.

07-89: **Function Definitions**

- Pseudo-Assembly for Function Definition:

```
<Label for start of function>
<Code to set up activation record>

<function body>

<label for end of function>
<code to clean up activation record>
<return>
```

07-90: **Creating AATs in Java**

- Labels
    - We will need to create (a potentially larger number of) assembly language labels
    - We need a way to create an arbitrarily large number of unique labels
    - We'd like them to have (reasonably) intuitive names
        - LABEL3011324 is not very meaningful

07-91: **Creating AATs in Java**

- Labels:

```
class Label {

    public Label() { ... }

    public Label(String s) { ... }

    public static AbsLabel(String s) { ... }

}
```

07-92: **Creating AATs in Java**

- Instead of traversing the Abstract Syntax Tree twice – once to do semantic analysis, and once to create Abstract Assembly Trees, we will traverse the AST only once
- Like to separate as much of the AAT building code as possible from the semantic analyzer

07-93: **Creating AATs in Java**

```
public class AATBuildTree {

public AATStatement functionDefinition(AATStatement body, int framesize,
                                       Label start, Label end)
public AATStatement ifStatement(AATExpression test, AATStatement ifbody,
                                AATStatement elsebody)
public AATExpression allocate(AATExpression size)
public AATStatement whileStatement(AATExpression test, AATStatement whilebody)
public AATStatement dowhileStatement(AATExpression test, AATStatement dowhilebody)
public AATStatement forStatement(AATStatement init, AATExpression test,
                                 AATStatement increment, AATStatement body)
public AATStatement emptyStatement()
public AATStatement callStatement(Vector actuals, Label name)
public AATStatement assignmentStatement(AATExpression lhs, AATExpression rhs)
public AATStatement sequentialStatement(AATStatement first, AATStatement second)
public AATExpression baseVariable(int offset)
public AATExpression arrayVariable(AATExpression base, AATExpression index,
                                   int elementSize)
public AATExpression classVariable(AATExpression base, int offset)
public AATExpression constantExpression(int value)
public AATExpression operatorExpression(AATExpression left, AATExpression right,
                                        int operator)
public AATExpression callExpression(Vector actuals, Label name)
public AATStatement returnStatement(AATExpression value, Label functionend)

}
```

07-94: **Creating AATs in Java**

- Using Interface: Examples

```
void foo(int a) {
   int b;
   int c;

   if (a > 2)
      b = 2;
   else
      c = a + 1;
}
```

07-95: **Creating AATs in Java**

- The AAT for the expression `(a > 2)` could be created with the code:

```
AATexpression e1;

e1 = bt.operatorExpression(bt.BaseVariable(4),
                           bt.ConstantExpression(2),
                           AATOperator.GREATER_THAN);
```

07-96: **Creating AATs in Java**

- The AAT for the statements `b=2;` and `c = a + 1;` could be created with the Java code:

```
AATstatement s1, s2;

s1 = bt.assignmentStatement(bt.baseVariable(0),
                            bt.constantExpression(2));
s2 = bt.assignmentStatement(bt.BaseVariable(-4),
                            bt.operatorExpression(
                                bt.baseVariable(4),
                                bt.constantExpression(1),
                                AATOperator.PLUS));
```

07-97: **Creating AATs in Java**

- The AAT for the statements `if (a > 2) b = 2; else c = a + 1;` could be created with the Java code:

```
AATstatement s3;

s3 = bt.ifStatement(e1,s1,s2);
```

07-98: **AAT Creation Interface**

- `AATstatement IfStatement(AATexpression test, AATstatement ifbody, AATstatement elsebody);`
  - **test:** An AAT, representing the test of the if statement
  - **ifbody:** An AAT, representing the "if" portion of the statement
  - **elsebody:** An AAT, representing the "else" portion of the statement (could be empty)

07-99: **AAT Creation Interface**

- `AATstatement FunctionDefintion(AATstatement body, int framesize, Label start Label end)`
  - **body:** The body of the function
  - **framesize:** The size of the frame devoted to local variables
  - **start:** The assembly language label for the start of the function
  - **end:** The assembly language label for use by return statements (to appear just before the cleanup code)

07-100: **AAT Creation Interface**

- `AATstatement ReturnStatement(AATexpression value, Label functionend);`
  - **value:** The value to return
  - **functionend:** The label to jump to, after the Result register has been set to `value`

07-101: **AAT Creation Interface**

- `public AATexpression Allocate(AATexpression size);`
  - Allocate creates an AAT that makes a call to the built-in function allocate, which takes as input the size (in bytes) to allocate, and returns a pointer to the beginning of the allocated block.
    - (More on *how* memory managers work later ...)

07-102: **Creating AATs**

- To create AATs:
  - Modify the semantic analyzer so that:
    - analyzeStatement returns an AATstatement
    - analyzeExpression returns both a Type (as before) *and* and AATexpression
    - Add calls to the interface to create AATs

07-103: **Creating AATs**

- analyzeExpression needs to return two pieces of data:
  - The type of the expression (to be used in type checking)
  - An AATexpression – assembly code for the expression

07-104: **Creating AATs**

- analyzeExpression needs to return two pieces of data:
  - The type of the expression (to be used in type checking)
  - An AATexpression – assembly code for the expression
- For the IntegerLiteral with value 3:
  - Return `IntegerType()` and `ConstantExpression(3)`

07-105: **Creating AATs**

- VisitOperatorExpression needs to return two pieces of data:
  - The type of the expression (to be used in type checking)
  - An AATexpression – assembly code for the expression
- For the sum of two expressions:
  - Return `IntegerType()` and `bt.operatorExpression(leftExp, rightExp, AAT_PLUS)`
    - Get values of `leftExp` and `rightExp` from calls to "accept" from left and right children.

07-106: **Creating AATs**

- Visitor functions can return any class
- Create a new class (internal to SemanticAnalyzer)

```
class TypeClass {
  public TypeClass(Type type, AATExpression value) {
    type_ = type;
    value_ = value;
  }
  public Type type() {
    return type_;
  }

  public AATExpression value() {
    return value_;
  }

  public void settype(Type type) {
    type_ = type;
  }

  public void setvalue(AATExpression value) {
    value_ = value;
  }

  private Type type_;
  private AATExpression value_;
}
```

07-107: **Creating AATs**

```
    public Object VisitOperatorExpression(ASTOperatorExpression opexpr) {
        TypeClass left = (TypeClass) opexpr.left().Accept(this);
        TypeClass right = (TypeClass) opexpr.right().Accept(this);
        int operator;
        AATExpression leftv = left.value();
        AATExpression rightv = right.value();
        Type leftt = left.type();
        Type rightt = right.type();


        if (opexpr.operator() == ASTOperatorExpression.PLUS) {
            if (leftt != IntegerType.instance() ||
                rightt != IntegerType.instance())
                e.error(opexpr.line(), "+ operator requires integer operands");
            return new TypeClass(IntegerType.instance(),
                                 bt.operatorExpression(leftv,rightv,
                                                       AATOperator.PLUS));
        } else ...  (more operators)

}
```

07-108: **Variables**

- To implement variables, we need to know the offset of each variable.
- Variable entries will contain an offset, as well as a type

(See `VariableEntry.java`, on other screen)

07-109: **Variable Declarations**

- Maintain a current offset
  - At the beginning of each function, set the offset to zero
  - When a variable is declared, use the current offset in the variable declaration, and decrement the offset by `WORDSIZE` (defined in MachineDependent.java)

07-110: **Variable Declarations**

- All variables are the same size (everything is either an integer, or a pointer – which is also an integer) – so managing offsets is (relatively) easy.
- If we had C-style structs on the stack, then we would need to calculate the size of each data structure to correctly determine offsets.

07-111: **Parameters**

- We will also need to calculate offsets for parameters
  - First parameter is at offset WORSIZE
  - Second parameter is at offset WORDSIZE*2
  - Third parameter is at offset WORDSIZE*3
  - ... etc

07-112: **Instance Var Declarations**

- Instance variables will also need to include an offset
  - Offset from the beginning of the data segment for the class
  - Offsets should be *negative* – base, base-WORDSIZE, base-2*WORDSIZE, etc. – just like variable declarations
- When analyzing a class definition:

- Set the offset to 0
- When an instance variable is declared, use the current offset, and decrement it by WORDSIZE

07-113: **Variables**

```
public Object VisitBaseVariable(ASTBaseVariable base) {
   VariableEntry var = variableEnv.find(base.name());
   if (var == null) {
      e.error(base.line(),"Variable " + base.name() +
              " is not defined in this scope");
      return new TypeClass(IntegerType.instance(), null);
   }  else {
      return new TypeClass(var.type(),
                           bt.baseVariable(var.offset()));
   }
}
```

07-114: **Project Hints**

- First, implement the AATBuildTree interface
- Test interface implementation
  - Some test files are provided
  - You will need to expand them
- Add AATBuildTree calls to semantic analyzer, to build the trees.
- Check to ensure that your trees are built correctly!
  - Code that produces a tree without segmentation faults $\neq$ working code! Go over the generated assembly trees by hand, to make sure they are correct
  - A pretty-printer for assembly trees has been provided

07-115: **Project Hints**

- *IF* your semantic analyzer is working perfectly, and is well coded, and you have a good understanding of all of the tree structures, then this project is slightly less time consuming than the semantic analyzer.
- However, it is still a relatively large project (easily the second hardest for this class), and you should allocate a good chunk of time to complete it.
- If all of the above conditions do not hold (especially if your semantic analyzer is not yet at 100%), allocate an *extra* chunk of time to complete this project.
- Only one more after this!