Compilers

CS414-2014-FR

Final Review

David Galles

Department of Computer Science
University of San Francisco

FR-0: Deterministic Finite Automata

- Set of states
- Initial State
- Final State(s)
- Transitions

DFA for else, end, identifiers

Combine DFA

FR-1: DFAs and Lexical Analyzers

- Given a DFA, it is easy to create C code to implement it
- DFAs are easier to understand than C code
  - Visual – almost like structure charts
- ... However, creating a DFA for a complete lexical analyzer is still complex

FR-2: Automatic Creation of DFAs

We'd like a tool:

- Describe the tokens in the language
- Automatically create DFA for tokens
- Then, automatically create C code that implements the DFA

We need a method for describing tokens

FR-3: Formal Languages

- **Alphabet** Σ: Set of all possible symbols (characters) in the input file
  - Think of Σ as the set of symbols on the keyboard
- **String** w: Sequence of symbols from an alphabet
- **String length** $|w|$ Number of characters in a string: |car| = 3, |abba| = 4
  - **Empty String** ϵ: String of length 0: $|\epsilon| = 0$
- **Formal Language**: Set of strings over an alphabet

Formal Language $\neq$ Programming language – Formal Language is only a set of strings.

FR-4: Formal Languages

Example formal languages:

- Integers $\{0, 23, 44, \ldots\}$
- Floating Point Numbers $\{3.4, 5.97, \ldots\}$
- Identifiers {foo, bar, . . .}

FR-5: Language Concatenation

- **Language Concatenation** Given two formal languages L_1 and L_2, the concatenation of L_1 and L_2, $L_1 L_2 = \{xy | x \in L_1, y \in L_2\}$

For example:
{fire, truck, car} {car, dog} =
{firecar, firedog, truckcar, truckdog, carcar, cardog}

FR-6: Kleene Closure

Given a formal language L:

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^1 &= L \\ L^2 &= LL \\ L^3 &= LLL \\ L^4 &= LLLL \end{aligned}$$

$$L^* = L^0 \bigcup L^1 \bigcup L^2 \bigcup \ldots \bigcup L^n \bigcup \ldots$$

FR-7: Regular Expressions

Regular expressions are use to describe formal languages over an alphabet Σ:

Regular Expression	Language
ϵ	$L[\epsilon] = \{\epsilon\}$
$\mathtt{a} \in \Sigma$	$L[\mathtt{a}] = \{\mathtt{a}\}$
(MR)	$L[MR] = L[M]L[R]$
$(M\|R)$	$L[(M\|R)] = L[M] \bigcup L[R]$
$(M*)$	$L[(M*)] = L[M]*$

FR-8: r.e. Precedence

From highest to Lowest:

Kleene Closure *

Concatenation

Alternation |

ab*c|e = (a(b*)c) | e

FR-9: Regular Expression Examples

(a|b)*	all strings over {a,b}
(0|1)*	binary integers (with leading zeroes)
a(a|b)*a	all strings over {a,b} that begin and end with a
(a|b)*aa(a|b)*	all strings over {a,b} that contain aa
b*(abb*)*(a|ϵ)	all strings over {a,b} that do not contain aa

FR-10: r.e. Shorthand

["a","b","c","d"]	=	(a|b|c|d)
["b"-"g"]	=	["b","e","f","g"] = (b|e|f|g)
["b"-"g,"M"-"O"]	=	["b","e","f","g","M","N","O"] = (b|e|f|g|M|
M?	=	(M | ϵ)
M+	=	MM*
"vc"	=	The string vc exactly

FR-11: r.e. Shorthand Examples

Regular Expression	Language
if	{if}
["a"-"z"]["0"-"9","a"-"z"]*	Set of legal identifiers
["0"-"9"]+	Set of integers (with leading zeroes
(["0"-"9"]+"."["0"-"9"]*)| (["0"-"9"]*"."["0"-"9"]+)	Set of real numbers

Parsing

- Once we have broken an input file into a sequence of tokens, the next step is to determine if that sequence of tokens forms a syntactically correct program – parsing
- We will use a tool to create a parser – just like we used lex to create a parser
- We need a way to describe syntactically correct programs
  - Context-Free Grammars

FR-13: Context-Free Grammars

- Set of Terminals (tokens)
- Set of Non-Terminals
- Set of Rules, each of the form: $<$Non-Terminal$>$ $\rightarrow$ $<$Terminals & Non-Terminals$>$
- Special Non-Terminal – Initial Symbol

Generating Strings with CFGs

- Start with the initial symbol
- Repeat:
  - Pick any non-terminal in the string
  - Replace that non-terminal with the right-hand side of some rule that has that non-terminal as a left-hand side

Until all elements in the string are terminals

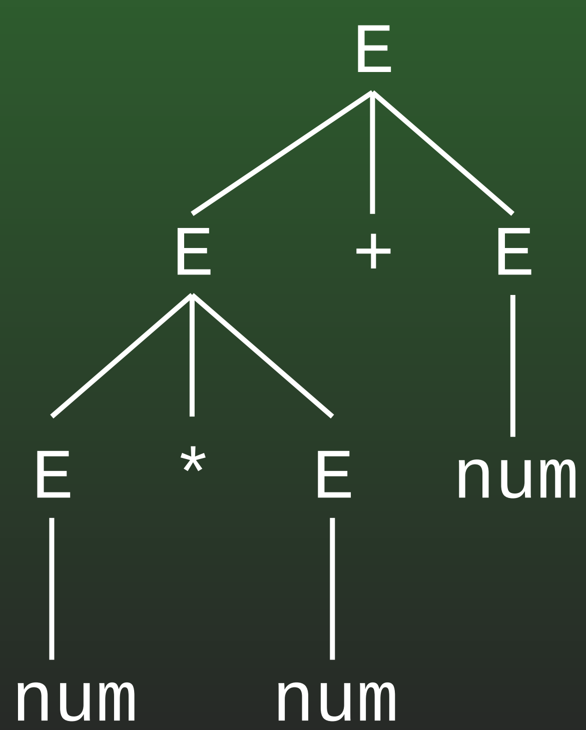
FR-15: CFG Example

$$E \longrightarrow E + E$$
$$E \longrightarrow E - E$$
$$E \longrightarrow E * E$$
$$E \longrightarrow E / E$$
$$E \longrightarrow \text{num}$$

FR-16: Parse Trees

A Parse Tree is a graphical representation of a derivation

$$\begin{aligned} E &\Rightarrow E + E \Rightarrow E * E + E \\ &\Rightarrow \text{num} * E + E \Rightarrow \text{num} * \text{num} + E \\ &\Rightarrow \text{num} * \text{num} + \text{num} \end{aligned}$$



FR-17: Ambiguous Grammars

- A Grammar is *ambiguous* if there is at least one string with more than one parse tree
- The expression grammar we've seen so far is ambiguous

$$E \rightarrow E + E$$
$$E \rightarrow E - E$$
$$E \rightarrow E * E$$
$$E \rightarrow E / E$$
$$E \rightarrow \text{num}$$

FR-18: Expression Grammar

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow \text{num}$

$F \rightarrow (E)$

FR-19: CFG for Statements

- Expressions: id, num
- Function calls: id(<input params>)
  - <input params> are expressions separated by commas
- Block Statements { < list of statements > }
- While statements (C syntax)

All statements are terminated by a semi-colon ;

FR-20: CFG for Statements

$S \rightarrow \text{id}(P);$

$S \rightarrow \{L\}$

$S \rightarrow \text{while } (E)\ S$

$E \rightarrow \text{id} \mid \text{num}$

$P \rightarrow \epsilon$

$P \rightarrow EP'$

$P' \rightarrow \epsilon$

$P' \rightarrow ,\ EP'$

$L \rightarrow \epsilon$

$L \rightarrow SL$

FR-21: LL(1) Parsers

These recursive descent parsers are also known as LL(1) parsers, for Left-to-right, Leftmost derivation, with 1 symbol lookahead

- The input file is read from left to right (starting with the first symbol in the input stream, and proceeding to the last symbol).
- The parser ensures that a string can be derived by the grammar by building a leftmost derivation.
- Which rule to apply at each step is decided upon after looking at just 1 symbol.

FR-22: Building LL(1) Parsers

$S' \rightarrow S$$

$S \rightarrow AB$

$S \rightarrow C$h

$A \rightarrow$ ef

$A \rightarrow \epsilon$

$B \rightarrow$hg

$C \rightarrow DD$

$C \rightarrow$ fi

$D \rightarrow$ g

ParseS use rule $S \rightarrow AB$ on e, h
use the rule $S \rightarrow C$h on f, g

FR-23: First sets

First(S) is the set of all terminals that can start strings derived from S (plus ϵ, if S can produce ϵ)

$S' \rightarrow S\$$	First(S') =
$S \rightarrow AB$	First(S) =
$S \rightarrow C$h	First(A) =
$A \rightarrow$ ef	First(B) =
$A \rightarrow \epsilon$	First(C) =
$B \rightarrow$hg	First(D) =
$C \rightarrow DD$	
$C \rightarrow$ fi	
$D \rightarrow$ g	

FR-24: First sets

First(S) is the set of all terminals that can start strings derived from S (plus ϵ, if S can produce ϵ)

$S' \rightarrow S\$$	First(S') = {e, f, g, h}
$S \rightarrow AB$	First(S) = {e, f, g, h}
$S \rightarrow C$h	First(A) = {e, ϵ}
$A \rightarrow$ ef	First(B) = {h}
$A \rightarrow \epsilon$	First(C) = {f, g}
$B \rightarrow$hg	First(D) = {g}
$C \rightarrow DD$	
$C \rightarrow$ fi	
$D \rightarrow$ g	

FR-25: Calculating First sets

- For each non-terminal S, set First(S) = {}
- For each rule of the form $S \rightarrow \gamma$, add First(γ) to First(S)
- Repeat until no changes are made

FR-26: Follow Sets

Follow(S) is the set of all terminals that can follow S in a (partial) derivation.

$S' \rightarrow S\$$	Follow(S') =
$S \rightarrow AB$	Follow(S) =
$S \rightarrow C$h	Follow(A) =
$A \rightarrow$ ef	Follow(B) =
$A \rightarrow \epsilon$	Follow(C) =
$B \rightarrow$hg	Follow(D) =
$C \rightarrow DD$	
$C \rightarrow$ fi	
$D \rightarrow$ g	

FR-27: Follow Sets

Follow(S) is the set of all terminals that can follow S in a (partial) derivation.

$S' \rightarrow S\$$	Follow(S') = { }
$S \rightarrow AB$	Follow(S) = {\$}
$S \rightarrow C$h	Follow(A) = {h}
$A \rightarrow$ ef	Follow(B) = {\$}
$A \rightarrow \epsilon$	Follow(C) = {h}
$B \rightarrow$hg	Follow(D) = {h, g}
$C \rightarrow DD$	
$C \rightarrow$ fi	
$D \rightarrow$ g	

FR-28: Calculating Follow sets

- For each non-terminal S, set Follow(S) = {}
- For each rule of the form $S \rightarrow \gamma$
  - For each non-terminal S_1 in γ, where γ is of the form $\alpha S_1 \beta$
    - If First(β) does not contain ϵ, add all elements of First(β) to Follow(S_1).
    - If First(β) does contain ϵ, add all elements of First(β) *except* ϵ to Follow(S_1), and add all elements of Follow(S) to Follow(S_1).
- If any changes were made, repeat.

Parse Tables

- Each row in a parse table is labeled with a non-terminal
- Each column in a parse table is labeled with a terminal
- Each element in a parse table is either empty, or contains a grammar rule
  - Rule $S \rightarrow \gamma$ goes in row S, in all columns of First(γ).
  - If First(γ) contains ϵ, then the rule $S \rightarrow \gamma$ goes in row S, in all columns of Follow(S).

FR-30: LL(1) Example

$E' \rightarrow E\$$

$E \rightarrow$ *E

$E \rightarrow R$

$R \rightarrow ST$

$T \rightarrow .ST$

$T \rightarrow \epsilon$

$S \rightarrow$ var$C$

$C \rightarrow$ [num] C

$C \rightarrow \epsilon$

FR-31: LL(1) Example

Non-Terminal	First	Follow
E'	*, var	
E	*, var	$
R	var	$
T	., ϵ	$
S	var	., $
C	[, ϵ	., $

FR-32: LL(1) Example

	*	.	var	[	num	]	$
E'	$E' \rightarrow E\$$		$E' \rightarrow E\$$				
E	$E \rightarrow$ *E		$E \rightarrow R$				
R			$R \rightarrow ST$				
T		$T \rightarrow$. ST					$T \rightarrow \epsilon$
S			$S \rightarrow$ var C				
C		$C \rightarrow \epsilon$		$C \rightarrow$ [num] C			$C \rightarrow \epsilon$

FR-33: Parsing

- LL(1) – Left-to-right, Leftmost derivation, 1-symbol lookahead parsers
  - Need to guess which rule to apply after looking at only the first element in the rule
- LR parsers – Left-to-right, Rightmost derivation parsers
  - Look at the entire right-hand side of the rule before deciding which rule to apply

FR-34: LR Parsing

- Maintain a stack
- Shift terminals from the input stream to the stack, until the top of the stack is the same as the right-hand side of a rule
- When the top of the stack is the same as the right-hand side of a rule *reduce* by that rule – replace the right-hand side of the rule on the stack with the left-hand side of the rule.
- Continue shifting elements and reducing by rules, until the input has been consumed and the stack contains only the initial symbol

FR-35: LR(0) Parsing

- Reads the input file Left-to-Right <u>L</u>R(0)
- Creates a Rightmost derivation L<u>R</u>(0)
- No Lookahead (0-symbol lookahead) LR(<u>0</u>)

LR(0) parsers are the simplest of the LR parsers

LR(0) Items

- An LR(0) item consists of
  - A rule from the CFG
  - A "." in the rule, which indicates where we currently are in the rule
- $S \rightarrow$ ab . c
  - Trying to parse the rule $S \rightarrow$ abc
  - Already seen "ab", looking for a "c"

FR-37: LR(0) States & Transitions

(0) $S' \rightarrow S\$$

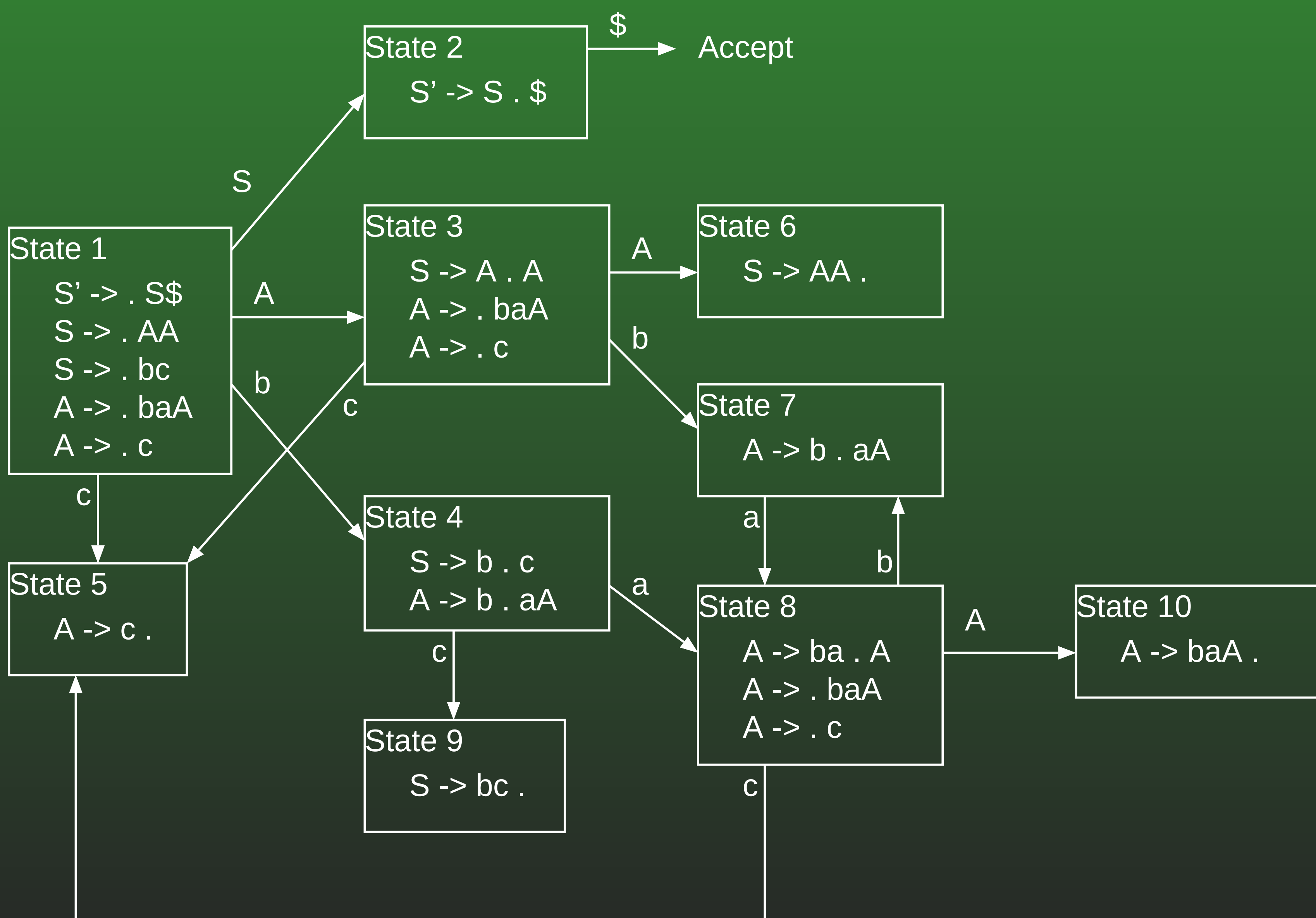
(1) $S \rightarrow AA$

(2) $S \rightarrow$ bc

(3) $S \rightarrow$ ba$A$

(4) $A \rightarrow$ c

FR-38: LR(0) States & Transitions



LR(0) Parse Table

	a	b	c	$	S	A
1		s4	s5		g2	
2				accept		
3		s7	s5			g6
4	s8		s9			
5	r(4)	r(4)	r(4)	r(4)		
6	r(1)	r(1)	r(1)	r(1)		
7	s8					
8		s7	s5			g10
9	r(2)	r(2)	r(2)	r(2)		
10	r(3)	r(3)	r(3)	r(3)		

Another LR(0) Example

(0) $S' \to S\$$

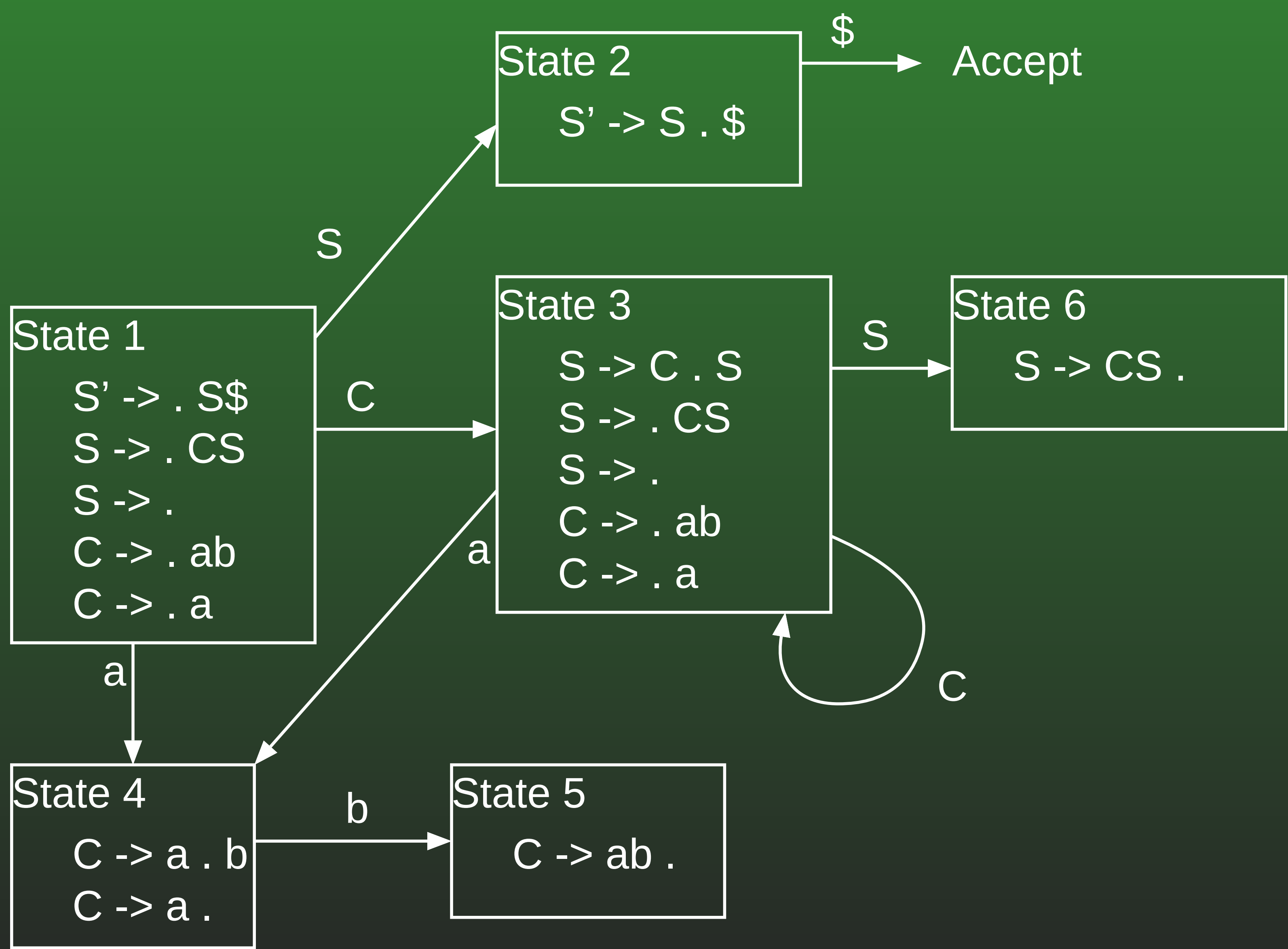
(1) $S \to CS$

(2) $S \to \epsilon$

(3) $C \to$ ab

(4) $C \to$ a

FR-41: LR(0) States & Transitions



LR(0) Parse Table

	a	b	\$	S'	S	C
1	s4,r(2)	r(2)	r(2)		g2	g3
2			accept			
3	s4,r(2)	r(2)	r(2)		g6	g3
4	r(4)	s5,r(4)	r(4)			
5	r(3)	r(3)	r(3)			
6	r(1)	r(1)	r(1)			

FR-43: SLR(1)

- Add simple lookahead (the *S* in SLR(1) is for *simple*
- In LR(0) parsers, if state k contains the item "$S \rightarrow \gamma$." (where $S \rightarrow \gamma$ is rule (n))
  - Put r(n) in state k, in all columns
- In SLR(0) parsers, if state k contains the item "$S \rightarrow \gamma$." (where $S \rightarrow \gamma$ is rule (n))
  - Put r(n) in state k, in all columns *in the follow set of* S

FR-44: SLR(1) Parse Table

	First	Follow
S’	a	
S	a	$
C	a	a, $

	a	b	$	S'	S	C
1	s4		r(2)		g2	g3
2			accept			
3	s4		r(2)		g6	g3
4	r(4)	s5	r(4)			
5	r(3)		r(3)			
6			r(1)			

LR(1) Items

- Like LR(0) items, contain a rule with a “.”
- Also contain lookahead information – the terminals that could follow this rule, *in the current derivation*
  - More sophisticated than SLR(1), which only look at what terminals could follow the LHS of the rule in *any* derivation

FR-46: LR(1) Example

(0) $S' \longrightarrow S$\$

(1) $S \longrightarrow AB$

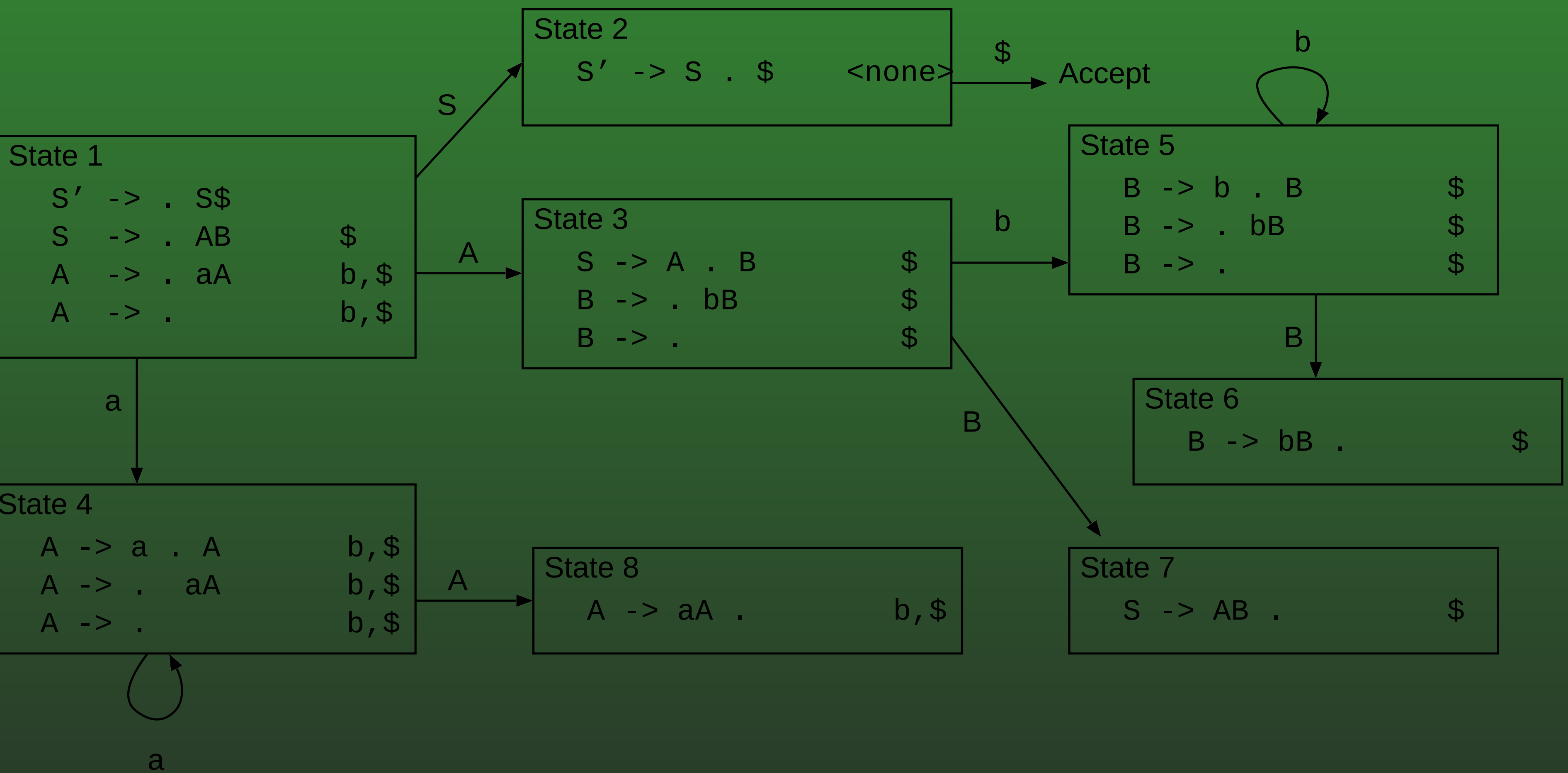
(2) $A \longrightarrow$ aA

(3) $A \longrightarrow \epsilon$

(4) $B \longrightarrow$ bB

(5) $B \longrightarrow \epsilon$

FR-47: LR(1) Example



LR(1) Example

	a	b	$	S'	S	A	B
1	s4	r(3)	r(3)		g2	g3	
2			accept				
3		s5	r(5)				g7
4	s4	r(3)	r(3)			g8	
5		s5	r(5)				g6
6			r(4)				
7			r(1)				
8		r(2)	r(2)				

FR-49: Abstract Syntax Tree (AST)

- Parse trees tell us exactly how a string was parsed
- Parse trees contain more information than we need
  - We only need the basic shape of the tree, not where every non-terminal is
  - Non-terminals are necessary for parsing, not for meaning
- An Abstract Syntax Tree is a simplifed version of a parse tree – basically a parse tree without non-terminals

FR-50: Parse Tree Example

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

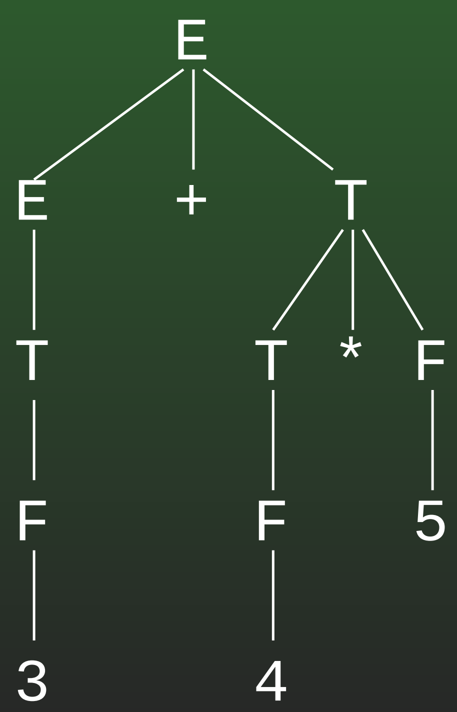
$F \rightarrow \text{num}$

Parse tree for 3 + 4 * 5

Parse Tree Example

$$E \rightarrow E + T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow F$$
$$F \rightarrow \text{num}$$

Parse tree for 3 + 4 * 5



FR-52: Abtract Syntax Tree Example

$E \rightarrow E + T$

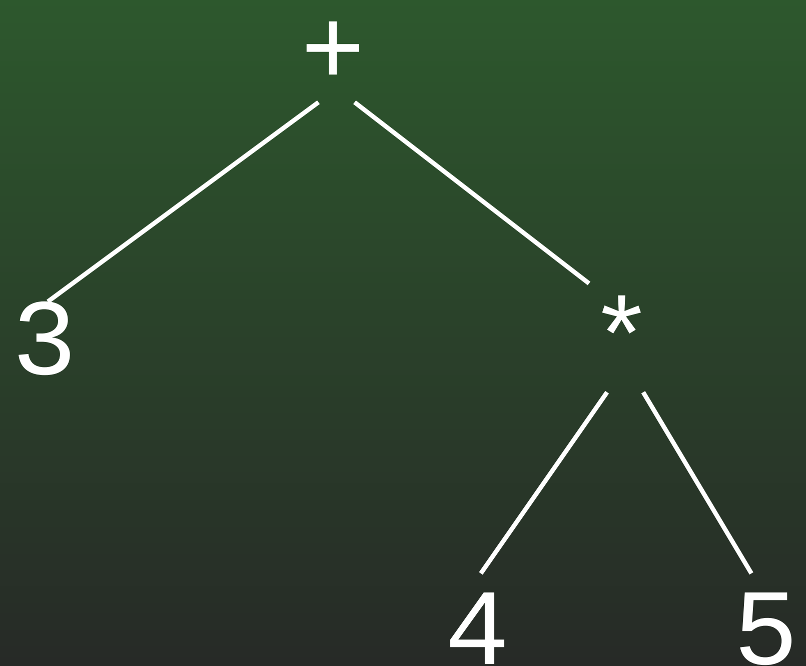
$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow$ num

Abstract Syntax Tree for 3 + 4 * 5



FR-53: AST – Expressions

- Simple expressions (such as integer literals) are a single node
- Binary expressions (+, *, /, etc.) are represented as a root (which stores the operation), and a left and right subtree
  - 5 + 6 * 7 + 8 (on whiteboard)

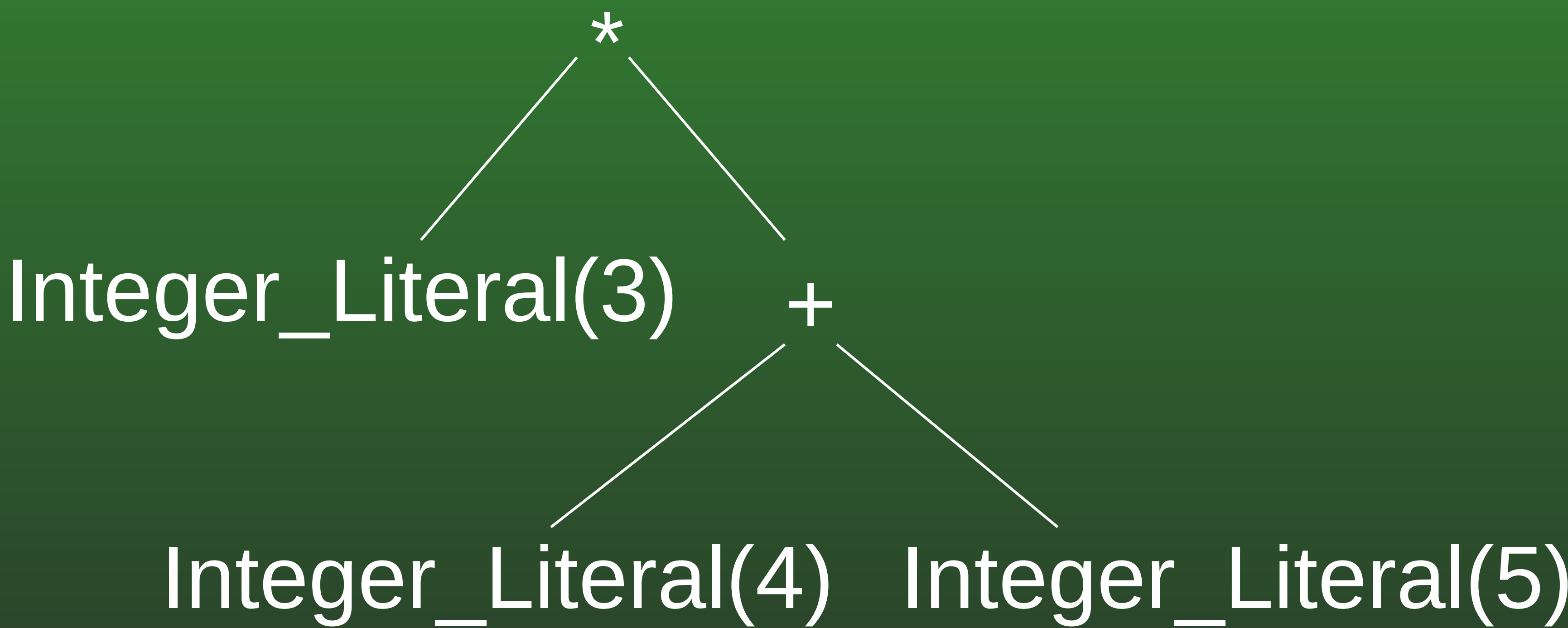
FR-54: AST – Expressions

- What about parentheses? Do we need to store them?
  - Parenthesis information is store in the shape of the tree
  - No extra information is necessary

3 * (4 + 5)

FR-55: AST – Expressions

3 * (4 + 5)



AST – Variables

- Simple variables (which we will call *Base Variables*) can be described by a single identifier.
- Instance variable accesses (`x.y`) require the name of the base variable (`x`), and the name of the instance variable (`y`).
- Array accesses (`A[3]`) require the base variable (`x`) and the array index (`3`).
- Variable accesses need to be extensible
  - `x.y[3].z`

FR-57: AST – Variables

- **Base Variables** Root is “BaseVar”, single child (name of the variable)
- **Class Instance Variables** Root is “ClassVar”, left subtree is the “base” of the variable, right subtree is the instance variable name
- **Array Variables** Root is “ArrayVar”, left subtree is the “base” of the variable, right subtree is the index

FR-58: # AST – Variables

FR-59: AST – Instance Variables

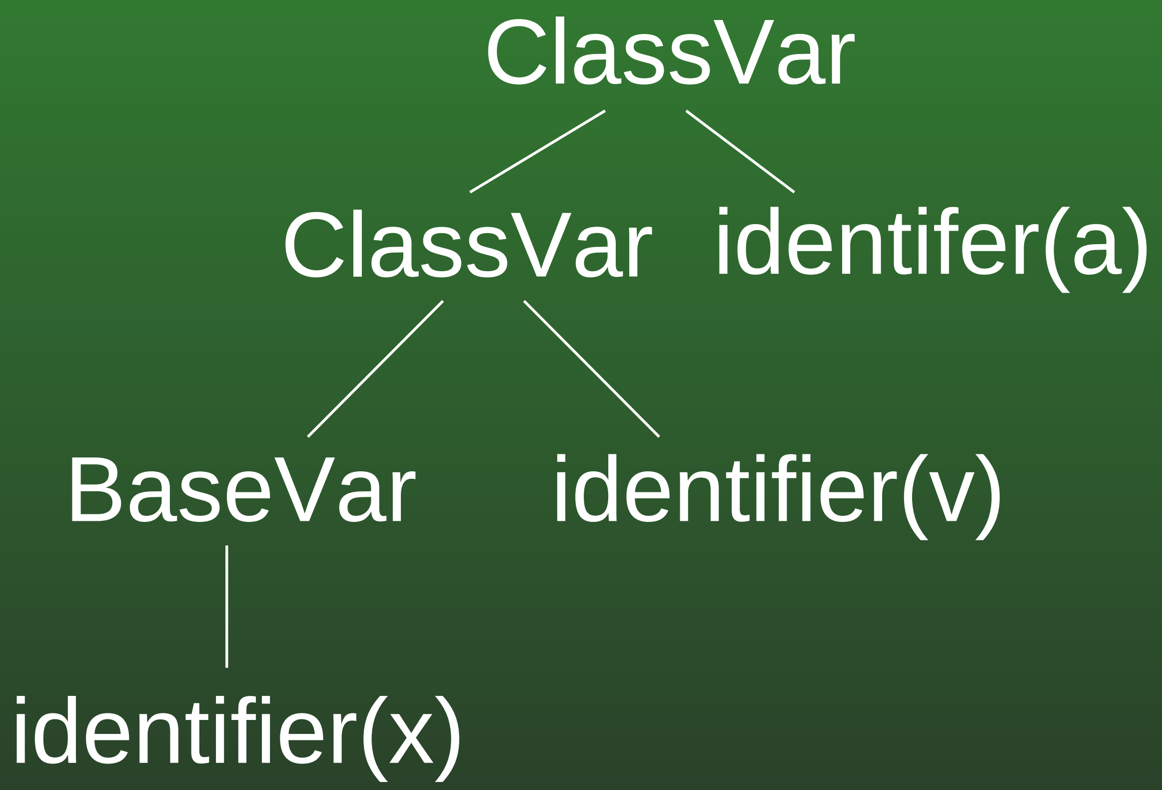
```
class simpleClass {
   int a;
   int b;
}
class complexClass {
   int u;
   simpleClass v;
}
void main() {
   complexClass x;
   x = new complexClass();
   x.v = new simpleClass();

   x.v.a = 3;
}
```

FR-60: AST – Instance Variables

x.v.a

FR-61: AST – Instance Variables

x.v.a

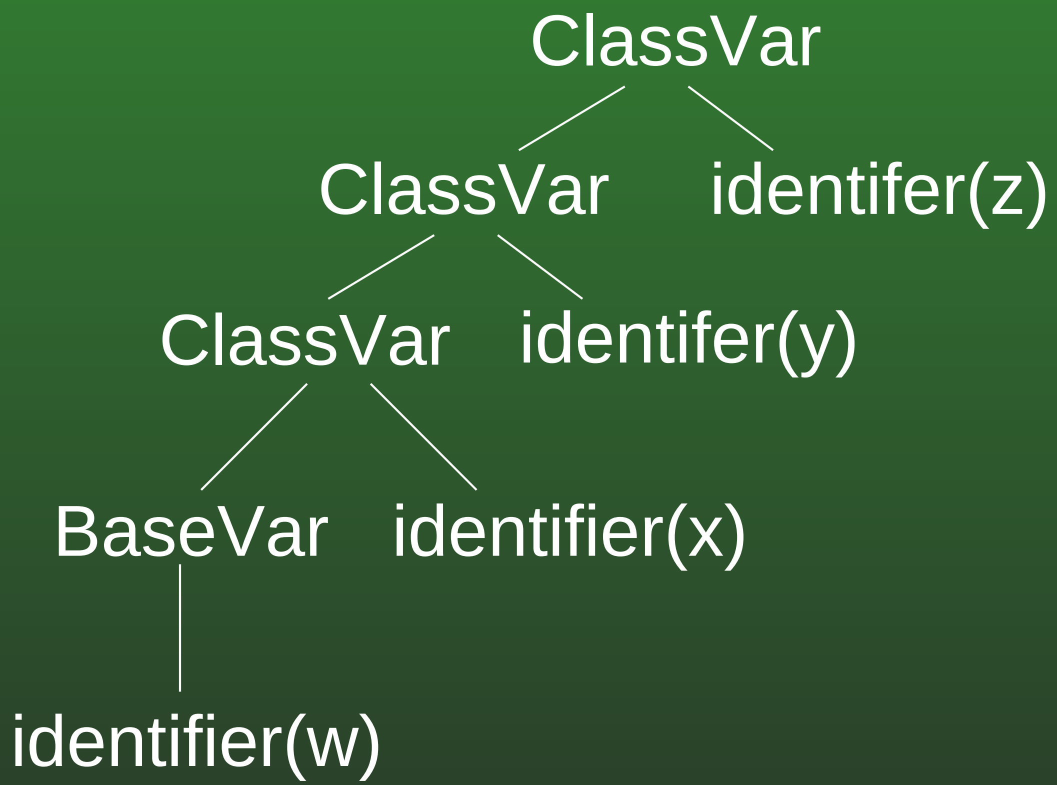


FR-62: AST – Instance Variables

w.x.y.z

FR-63: AST – Instance Variables

w.x.y.z

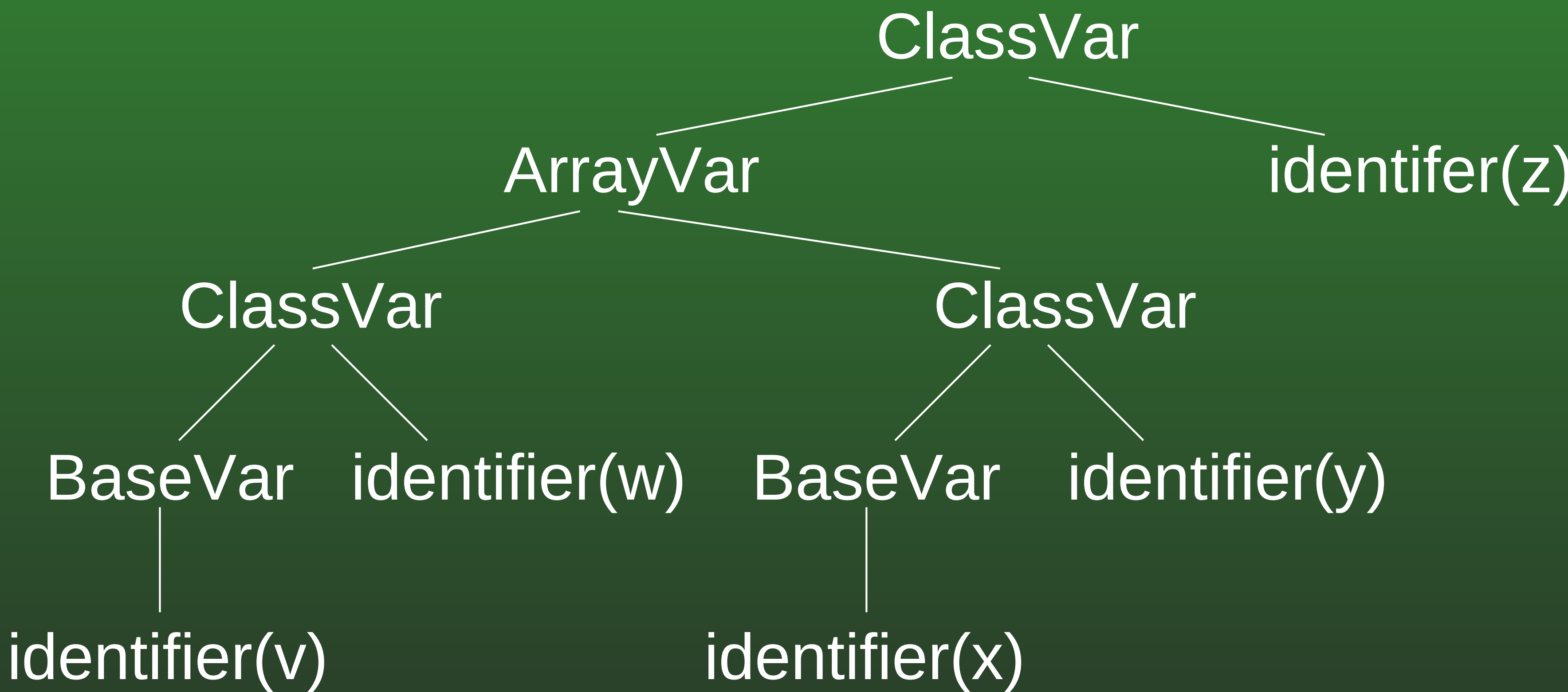


FR-64: AST – Instance Variables

v.w[x.y].z

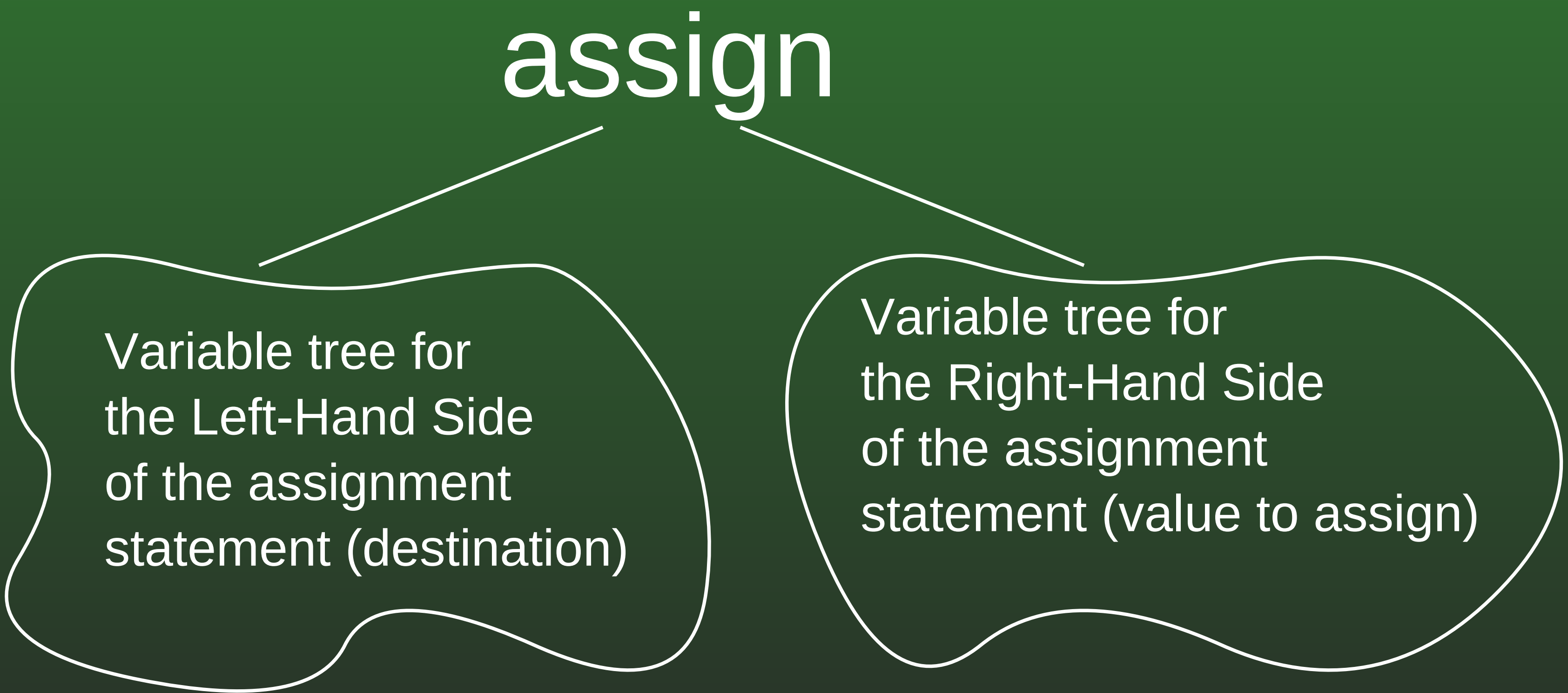
FR-65: AST – Instance Variables

v.w[x.y].z



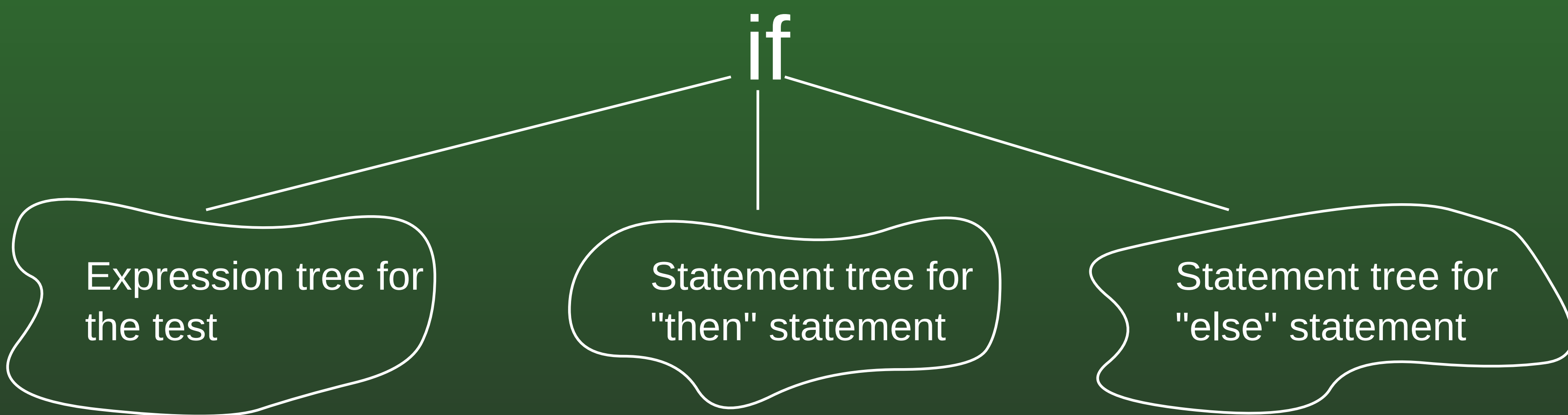
FR-66: AST – Statements

- **Assignment Statement** Root is “Assign”, children for left-hand side of assignment statement, and right-hand side of assignment statement



FR-67: AST – Statements

- **If Statement** Root is “If”, children for test, “then” clause, and “else” clause of the statement. The “else” tree may be empty, if the statement has no else.



AST – Statements

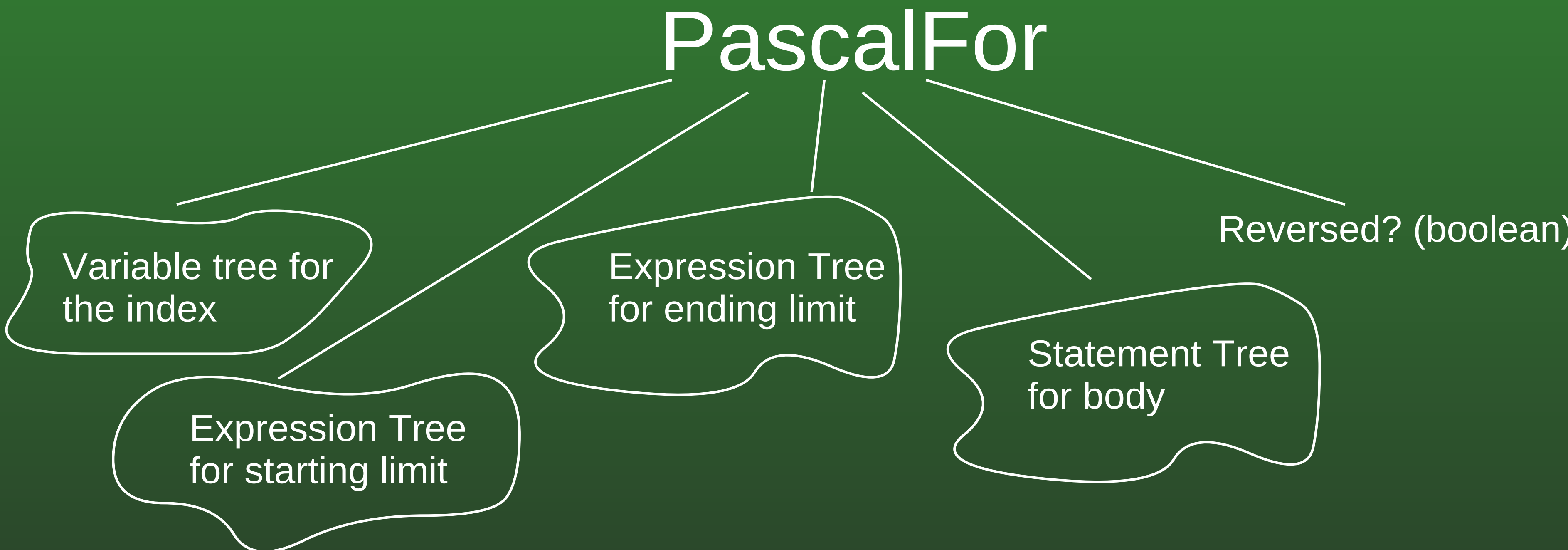
- Pascal-style for loops:

```
for i = 0 to 10 do  ...

for j = 10 downto 3 do ...
```

- If we are designing a new AST node for Pascal-style for loops, what should it look like? How many children? What should they be?

FR-69: AST – Statements

- Pascal-style for loops:



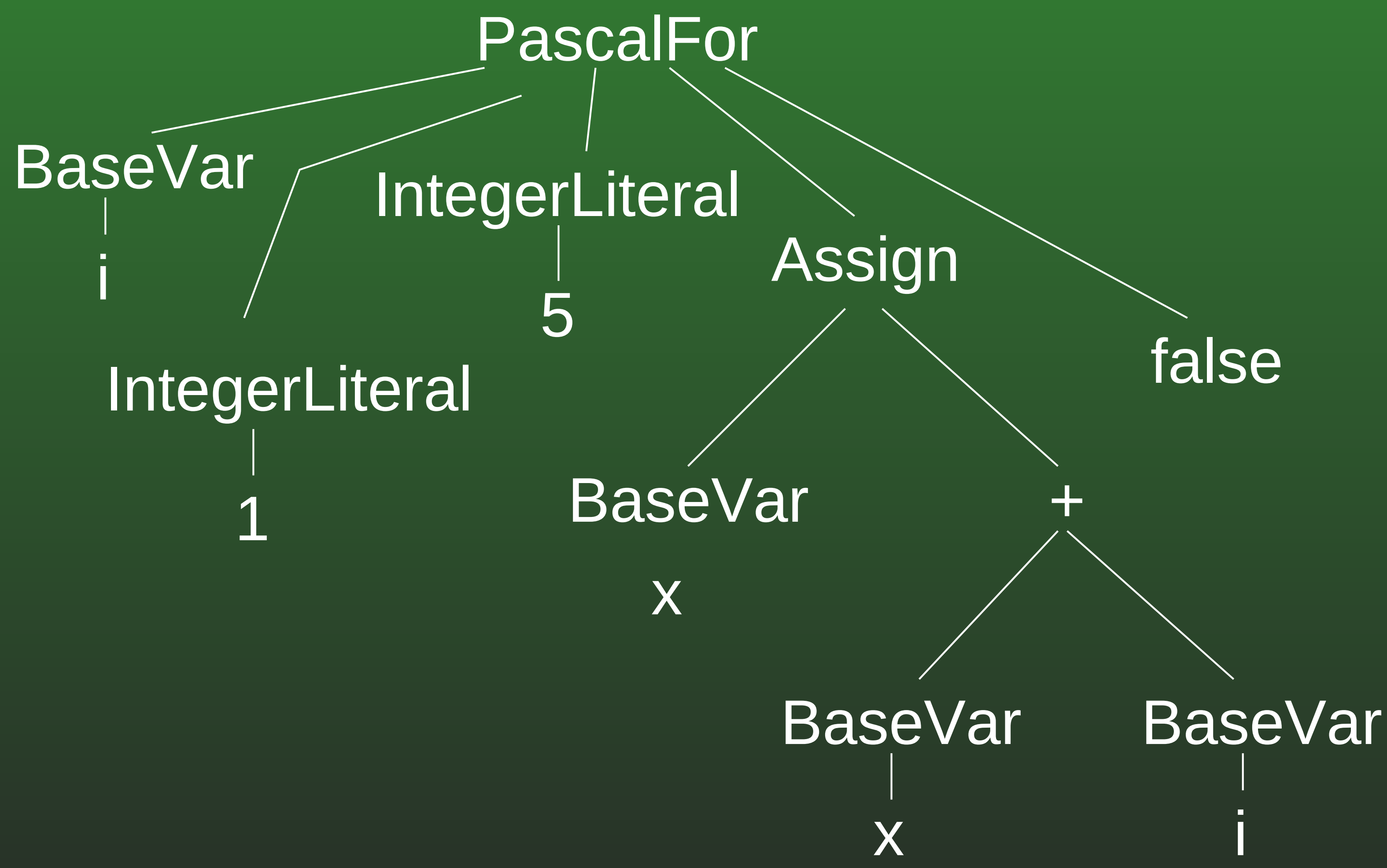
FR-70: AST – Statements

- Pascal-style for loops:

```
for i = 1 to 5 do x = x + i;
```

FR-71: AST – Statements

```
for i = 1 to 5 do x = x + i;
```



FR-72: Syntax / Semantic Errors

- A program has *syntax* errors if it cannot be generated from the Context Free Grammar which describes the language
- The following code has no *syntax* errors, though it has plenty of *semantic* errors:

```
void main() {
    if (3 + x - true)
        x.y.z[3] = foo(z)
}
```

- Why don't we write a CFG for the language, so that all syntactically correct programs also contain no semantic errors?

FR-73: Syntax / Semantic Errors

- Why don't we write a CFG for the language, so that all syntactically correct programs also contain no semantic errors?
- In general, we can't!
  - In simpleJava, variables need to be declared before they are used
  - The following CFG:
    - $L = \{ww | w \in \{a, b\}\}$

    is *not* Context-Free – if we can't generate this string from a CFG, we certainly can't generate a simpleJava program where all variables are declared before they are used.

FR-74: Semantic Errors

- Semantic Errors can be classified into the following broad categories:
- **Definition Errors**
- Most strongly typed languages require variables, functions, and types to be defined before they are used with some exceptions –
  - Implicit variable declarations in Fortran
  - Implicit function definitions in C

FR-75: Semantic Errors

- Semantic Errors can be classified into the following broad categories:
- **Structured Variable Errors**
  - x.y = A[3]
    - x needs to be a class variable, which has an instance variable y
    - A needs to be an array variable
  - x.y[z].w = 4
  - x needs to be a class variable, which has an instance variable y, which is an array of class variables that have an instance variable w

FR-76: Semantic Errors

- Semantic Errors can be classified into the following broad categories:
  - **Function and Method Errors**
    - foo(3, true, 8)
      - foo must be a function which takes 3 parameters:
      - integer
      - boolean
      - integer

FR-77: Semantic Errors

- Semantic Errors can be classified into the following broad categories:
  - **Type Errors**
  - Build-in functions – /, *, ||, &&, etc. – need to be called with the correct types
    - In simpleJava, +, -, *, / all take integers
    - In simpleJava, || &&, ! take booleans
    - Standard Java has polymorphic functions & type coercion

FR-78: Semantic Errors

- Semantic Errors can be classified into the following broad categories:
  - **Type Errors**
  - Assignment statements must have compatible types
  - When are types compatible?

FR-79: Semantic Errors

- Semantic Errors can be classified into the following broad categories:
  - **Type Errors**
  - Assignment statements must have compatible types
    - In Pascal, only *Identical* types are compatible

FR-80: Semantic Errors

- Semantic Errors can be classified into the following broad categories:
  - **Type Errors**
  - Assignment statements must have compatible types
    - In C, types must have the same structure
    - Coerceable types also apply

```
struct {                struct {
   int x;                  int z;
   char y;                 char x;
} var1;                 } var2;
```

FR-81: Semantic Errors

- Semantic Errors can be classified into the following broad categories:
  - **Type Errors**
  - Assignment statements must have compatible types
    - In Object oriented languages, can assign superclass value to a subclass variable

Semantic Errors

- Semantic Errors can be classified into the following broad categories:
  - **Access Violation Errors**
  - Accessing private / protected methods / variables
  - Accessing local functions in block structured languages
  - Separate files (C)

FR-83: Environment

- Much of the work in semantic analysis is managing environments
- Environments store current definitions:
  - Names (and structures) of types
  - Names (and types) of variables
  - Names (and return types, and number and types of parameters) of functions
- As variables (functions, types, etc) are declared, they are added to the environment. When a variable (function, type, etc) is accessed, its definition in the environment is checked.

FR-84: Implementing Environments

- Environments are implemented with Symbol Tables
- Symbol Table ADT:
  - Begin a new scope.
  - Add a key / value pair to the symbol table
  - Look up a value given a key. If there are two elements in the table with the same key, return the most recently entered value.
  - End the current scope. Remove all key / value pairs added since the last begin scope command

FR-85: Implementing Symbol Tables

- Implement a Symbol Table as an open hash table
  - Maintain an array of lists, instead of just one
  - Store (key/value) pair in the front of `list[hash(key)]`, where `hash` is a function that converts a key into an index
  - If:
    - The hash function distributes the keys evenly throughout the range of indices for the list
    - # number of lists = Θ(# of key/value pairs)

  Then inserting and finding take time $\Theta(1)$

FR-86: Implementing Symbol Tables

- What about `beginScope` and `endScope`?
- The key/value pairs are distributed across several lists – how do we know which key/value pairs to remove on an `endScope`?
  - If we knew exactly which variables were inserted since the last `beginScope` command, we could delete them from the hash table
  - If we always enter and remove key/value pairs from the beginning of the appropriate list, we will remove the correct items from the environment when duplicate keys occur.
  - How can we keep track of which keys have been added since the last beginScope?

FR-87: Implementing Symbol Tables

- How can we keep track of which keys have been added since the last beginScope?
- Maintain an auxiliary stack
  - When a key/value pair is added to the hash table, push the key on the top of the stack.
  - When a “Begin Scope” command is issued, push a special begin scope symbol on the stack.
  - When an “End scope” command is issued, pop keys off the stack, removing them from the hash table, until the begin scope symbol is popped

FR-88: 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

FR-89: 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)

FR-90: 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

FR-91: Stack Implementation

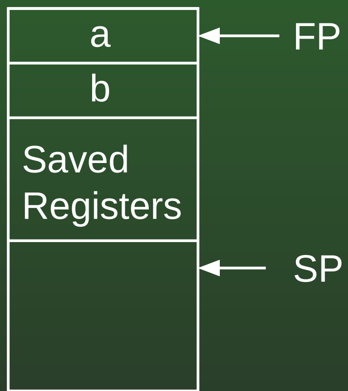
- Stack is implemented with two registers:
  - Frame Pointer (FP) points to the beginning of the current stack. frame
  - Stack Pointer (SP) points to the next free address on the stack.
- Stacks grow from large addresses to small addresses.

FR-92: 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`

FR-93: Stack Frames

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

    /* body of foo */
}
```

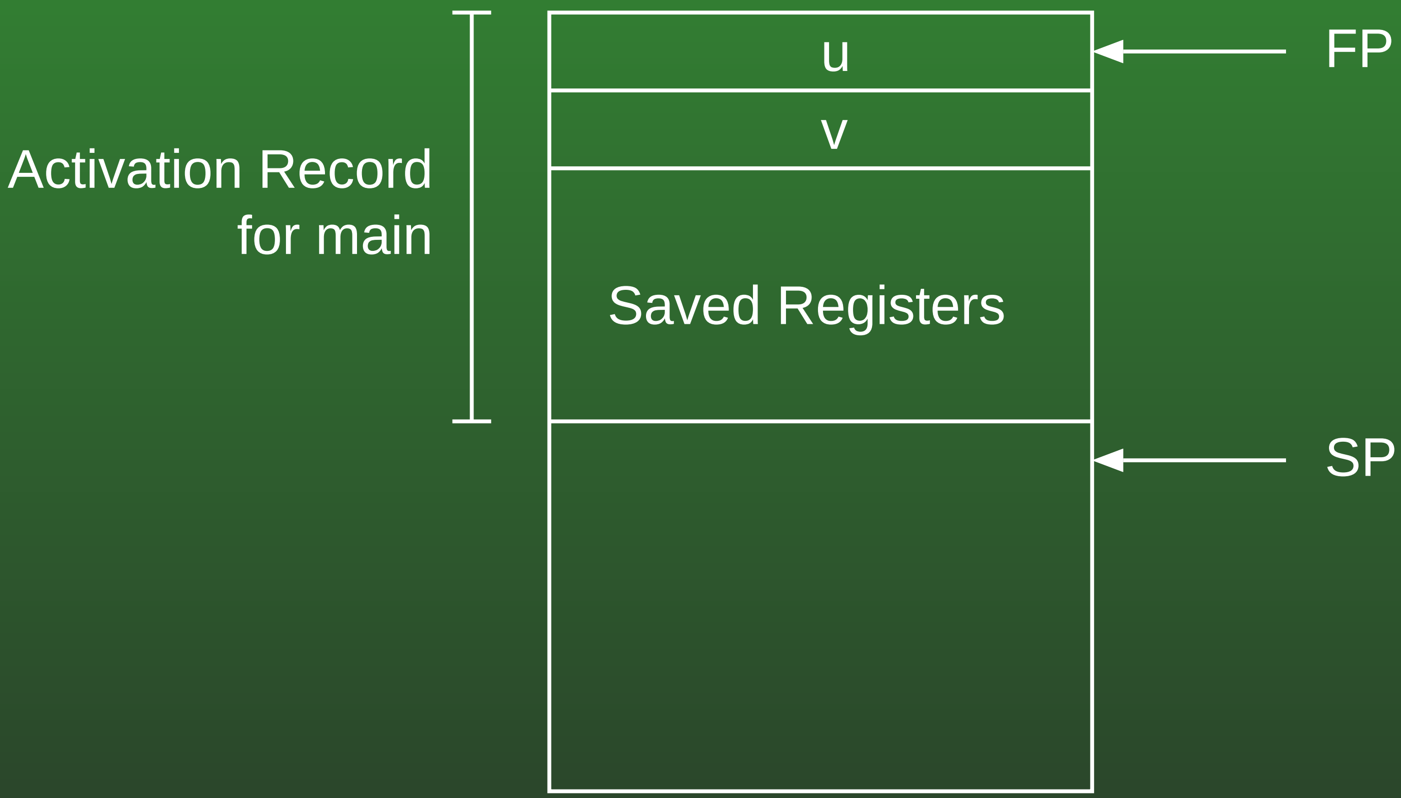


FR-94: Stack Frames

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

void main() {
   int u;
   int v;

   /* Label A */

   bar(1,2);
}

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

   foo(3,4);
}
```

```
int foo(int c, int d) {
   int y;
   int z;

   /* Label B */
}
```

FR-95: Stack Frames

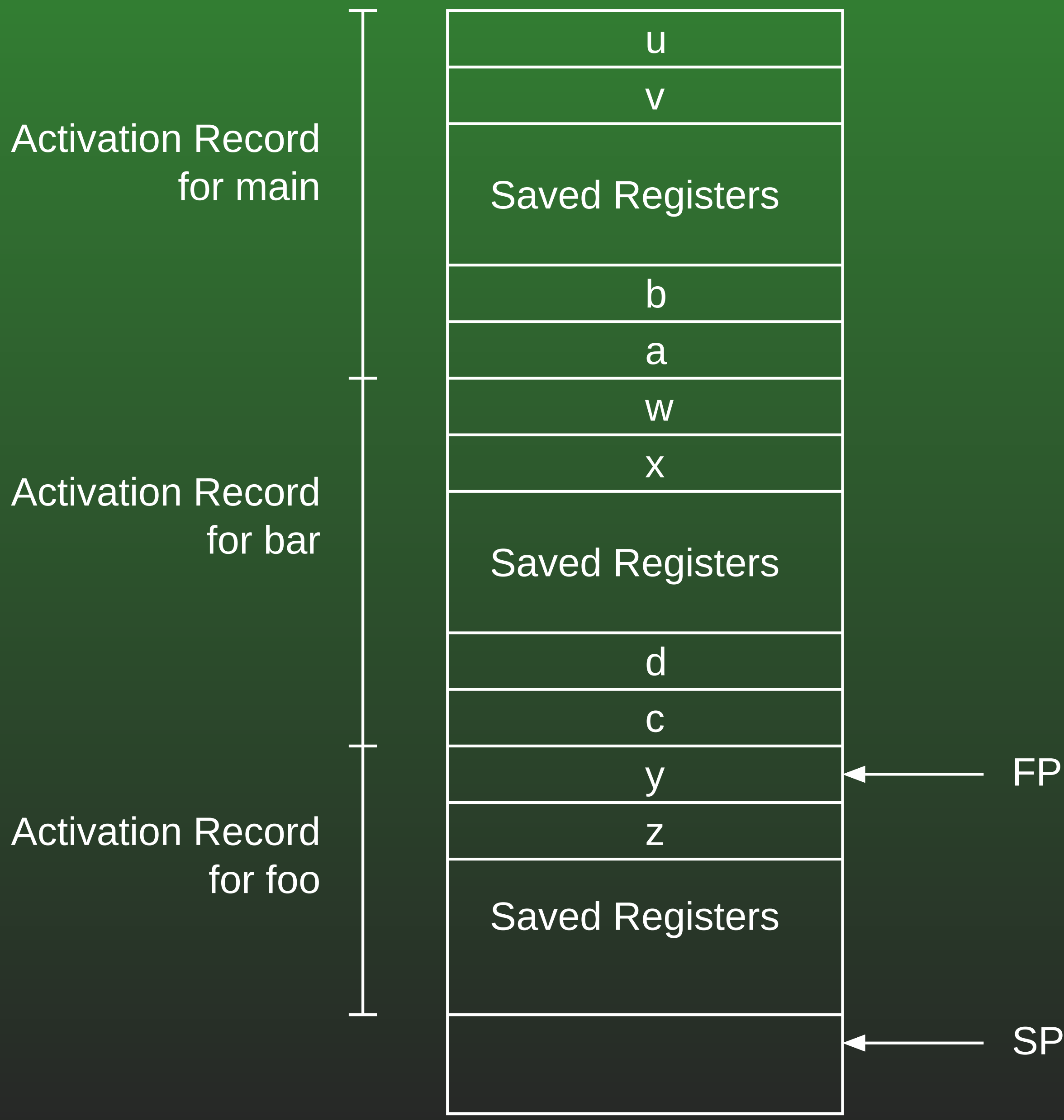


FR-96: Stack Frames

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

void main() {
   int u;
   int v;

   /* Label A */

   bar(1,2);
}

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

   foo(3,4);
}
```

```
int foo(int c, int d) {
   int y;
   int z;

   /* Label B */
}
```

FR-97: Stack Frames



FR-98: 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

FR-99: 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)

FR-100: Abstract Assembly

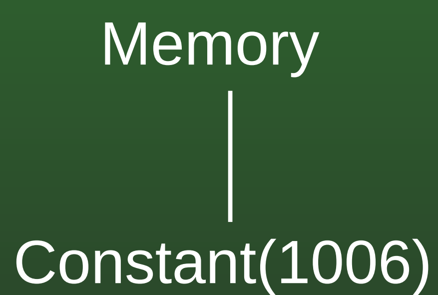
- There are two kinds of Assembly Trees:
  - Expression Trees, which represent values
  - Statement Trees, which represent actions
- Just like Abstract Syntax Trees

FR-101: Expression Trees

- Constant Expressions
  - Stores the value of the constant.
  - Only integer constants
    - (booleans are represented as integers, just as in C)

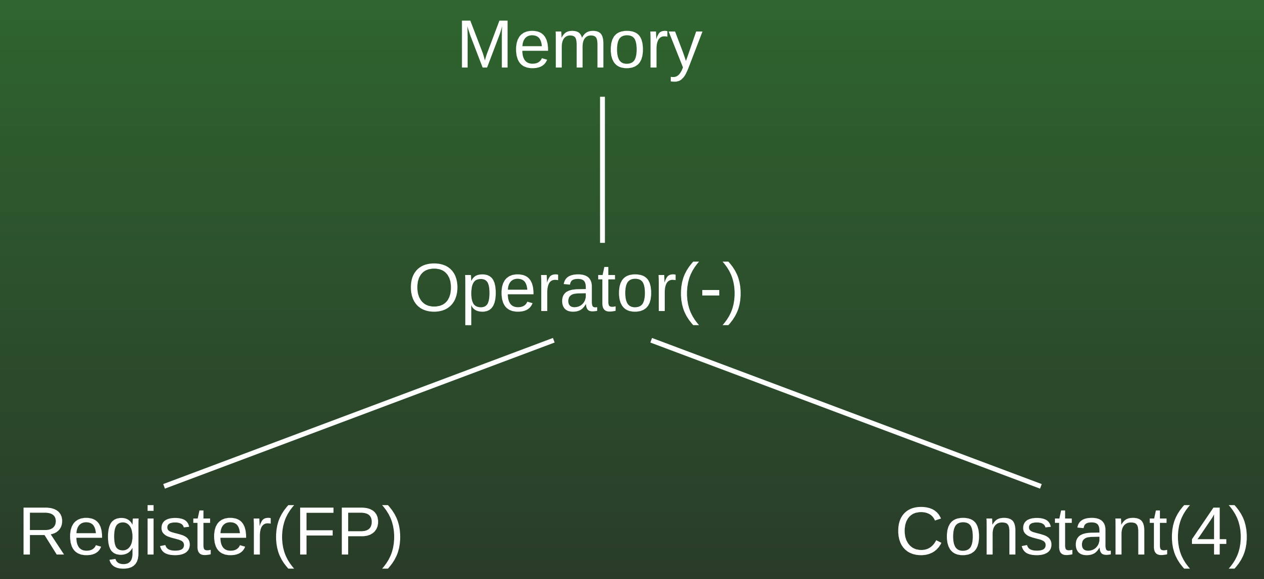
FR-102: Expression Trees

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



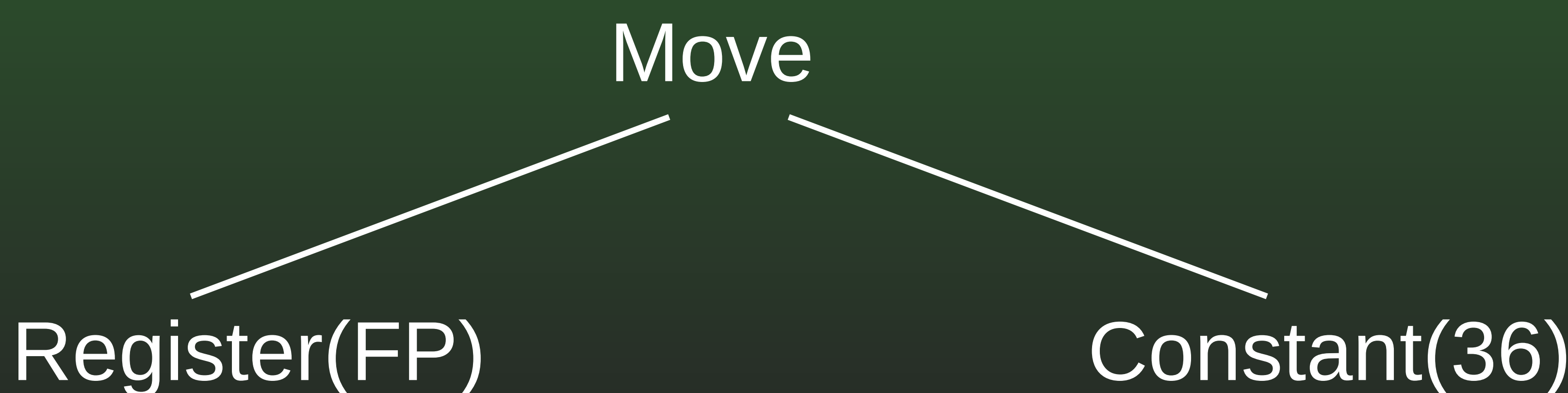
FR-103: Expression Trees

- Memory Expression
  - Represents a memory dereference. Contains the memory location to examine.
  - Local variable with an offset of 4 off the FP is



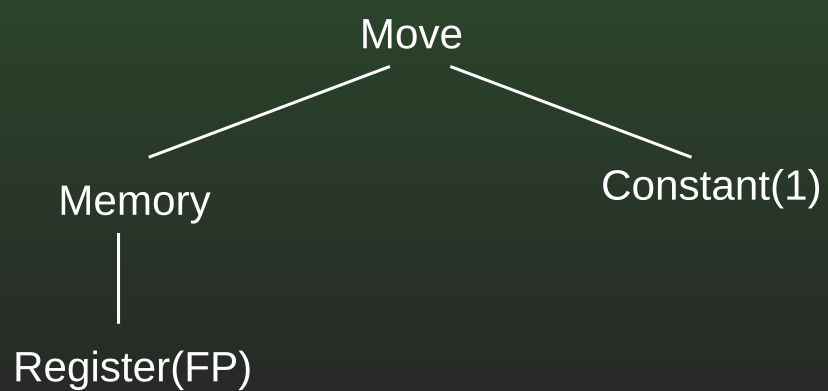
FR-104: Statement Trees

- 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:



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:

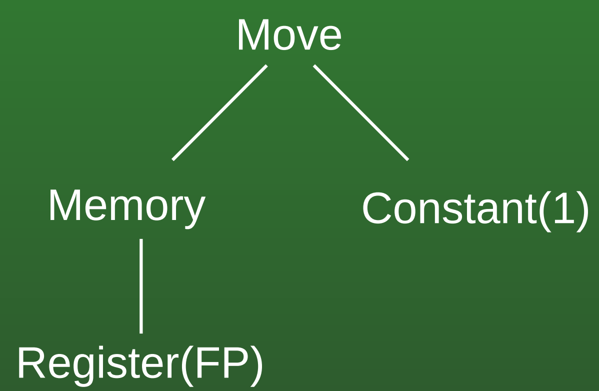


FR-106: Abstract Assembly Examples

```
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;
}
```

FR-107: Abstract Assembly Examples

```
x = 1;
```

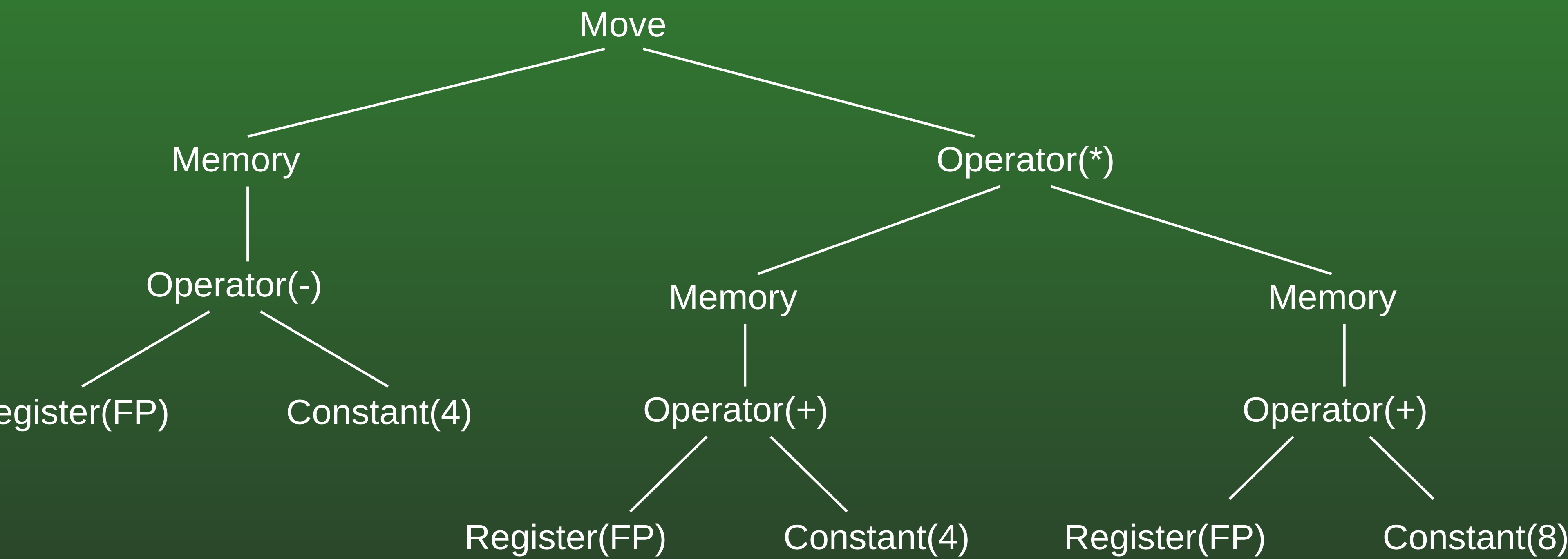


FR-108: Abstract Assembly Examples

```
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;
}
```

FR-109: Abstract Assembly Examples

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

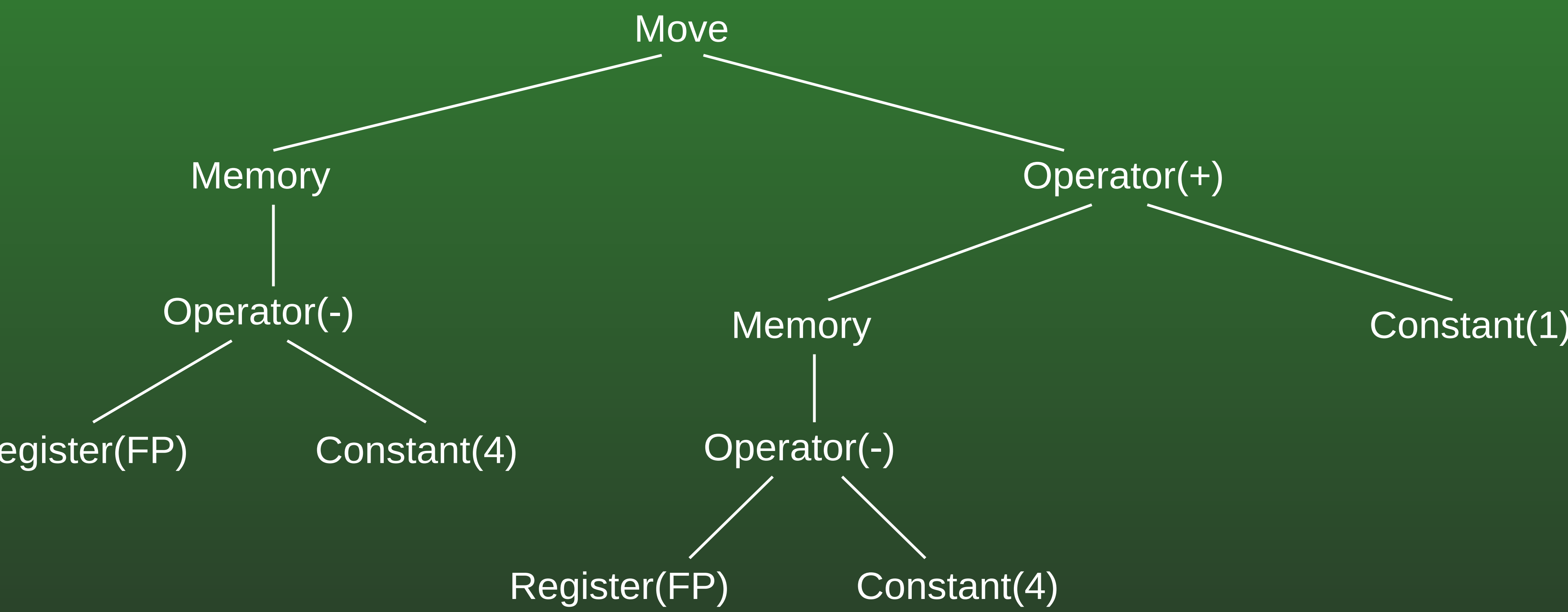


FR-110: Abstract Assembly Examples

```
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;
}
```

Abstract Assembly Examples

y++;

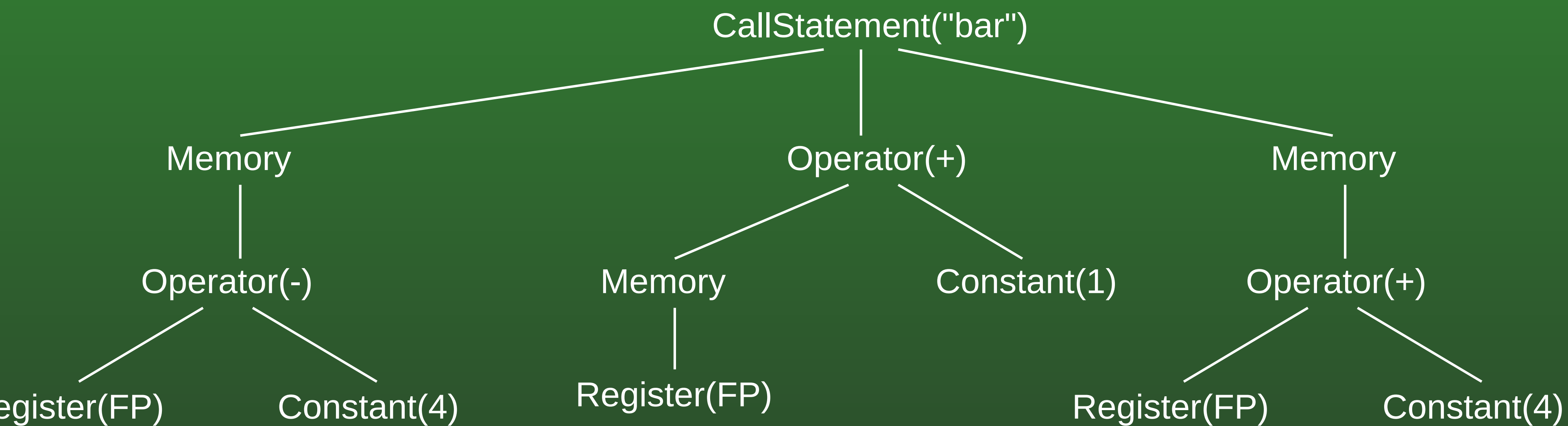


FR-112: Abstract Assembly Examples

```
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;
}
```

Abstract Assembly Examples

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

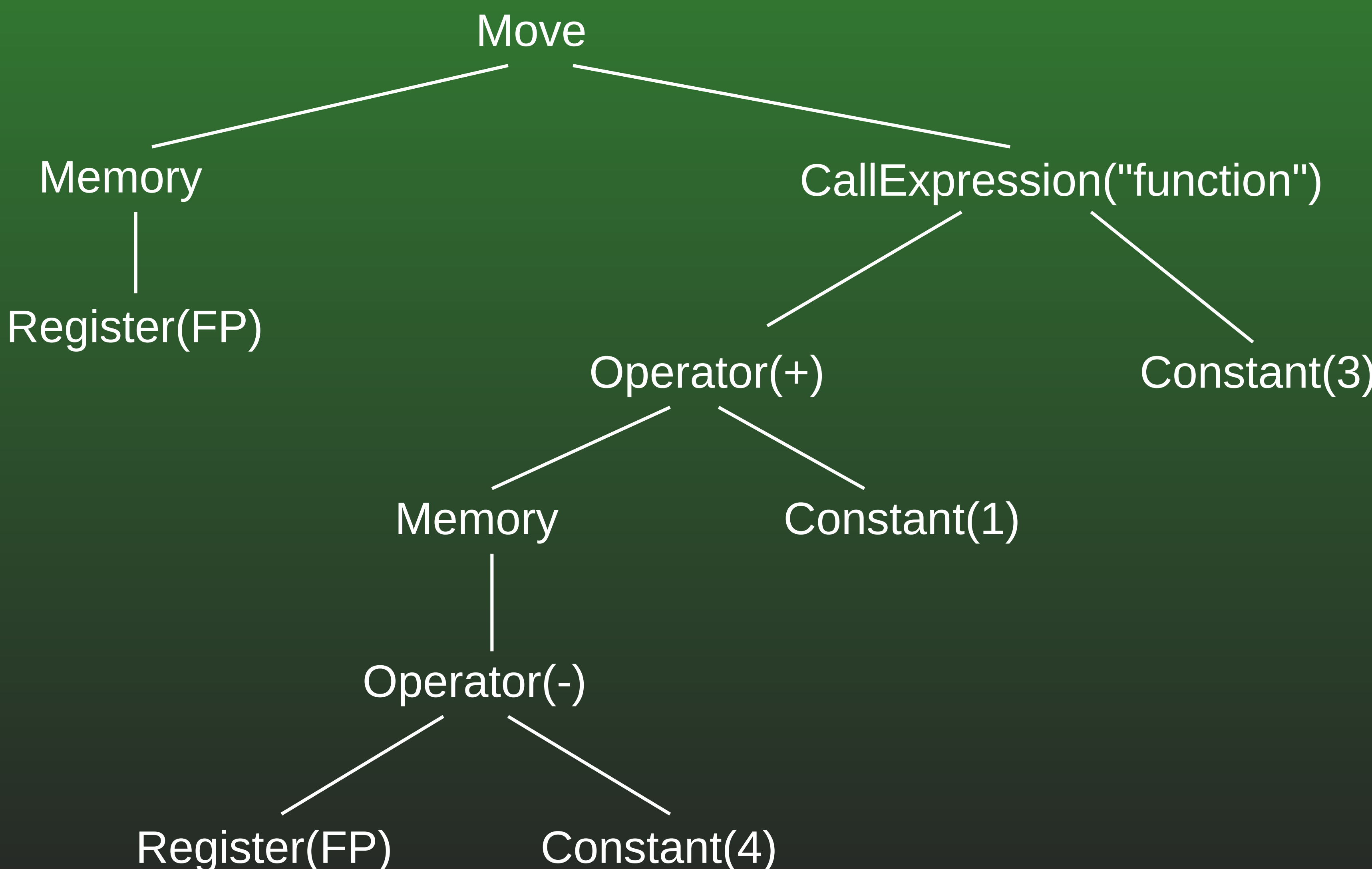


Abstract Assembly Examples

```
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;
}
```

FR-115: Abstract Assembly Examples

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

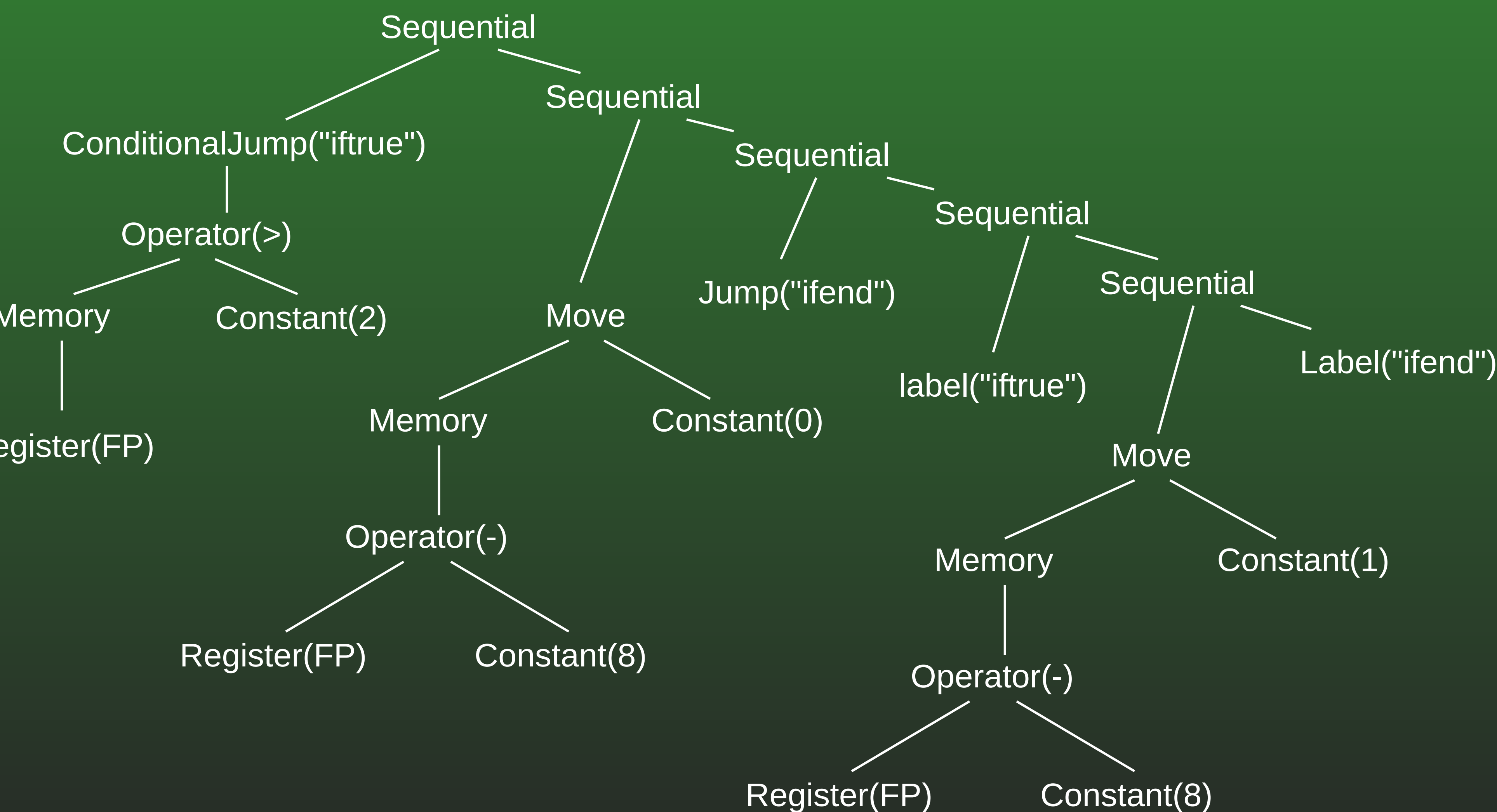


FR-116: Abstract Assembly Examples

```
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;              -|
}
```

Abstract Assembly Examples

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

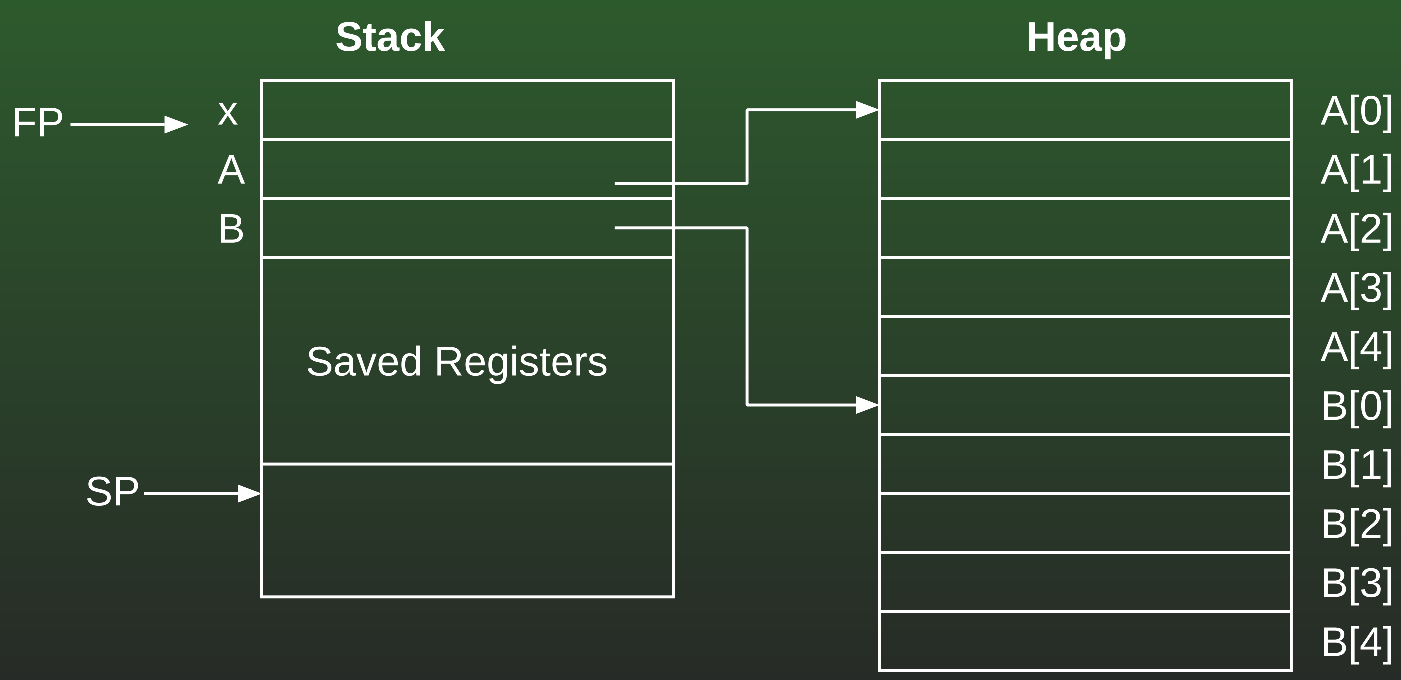


FR-118: Creating Abstract Assembly

- 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

FR-119: Array Variables

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



FR-120: Array Variables

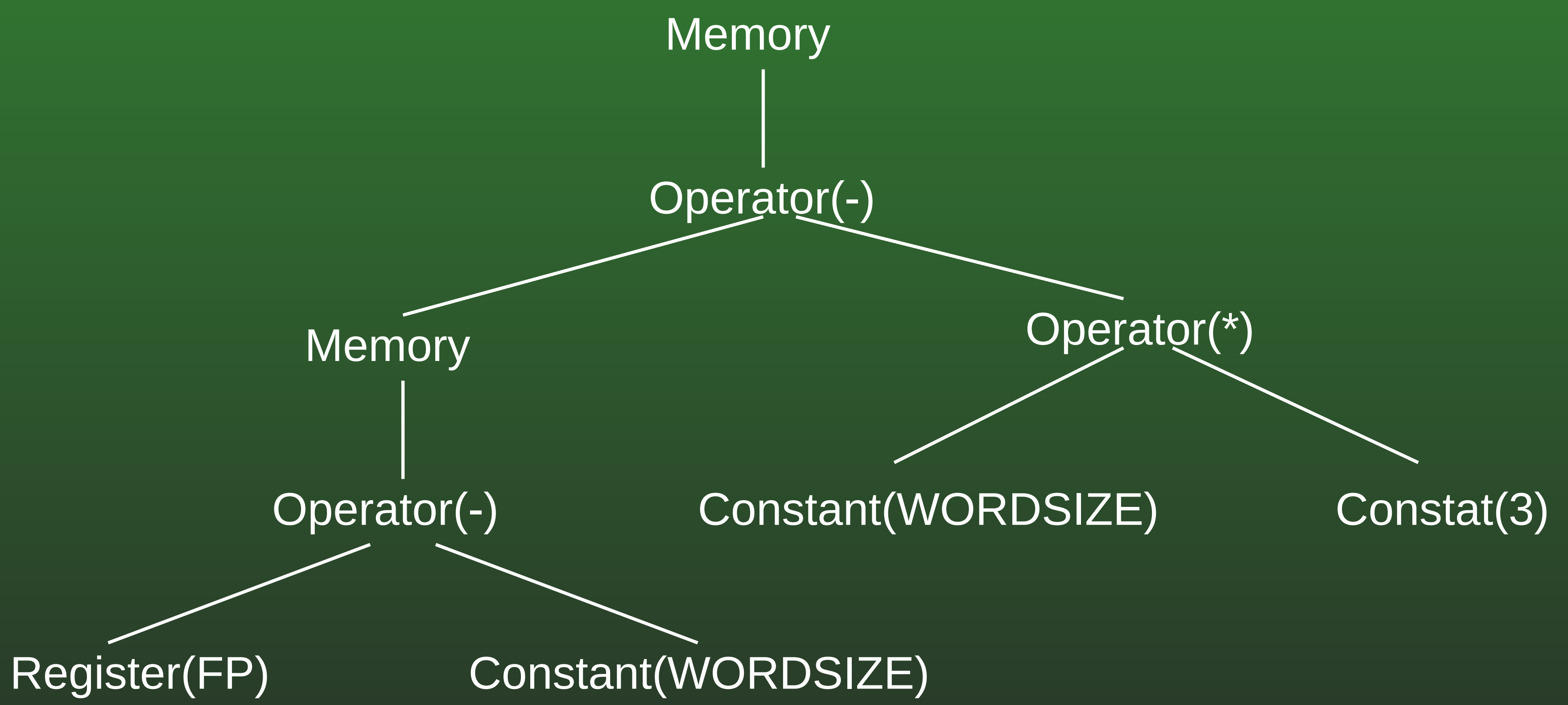
- How do we represent A[3] in abstract assembly?
  - Use the offset of `A` to get at the beginning of the array
  - Subtract 3 * (element size) from this pointer
    - In simpleJava, all variables take a single word. Complex variables – classes and arrays – are pointers, which also take a single word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

FR-121: Array Variables

- `A[3]`

FR-122: Array Variables

- `A[3]`

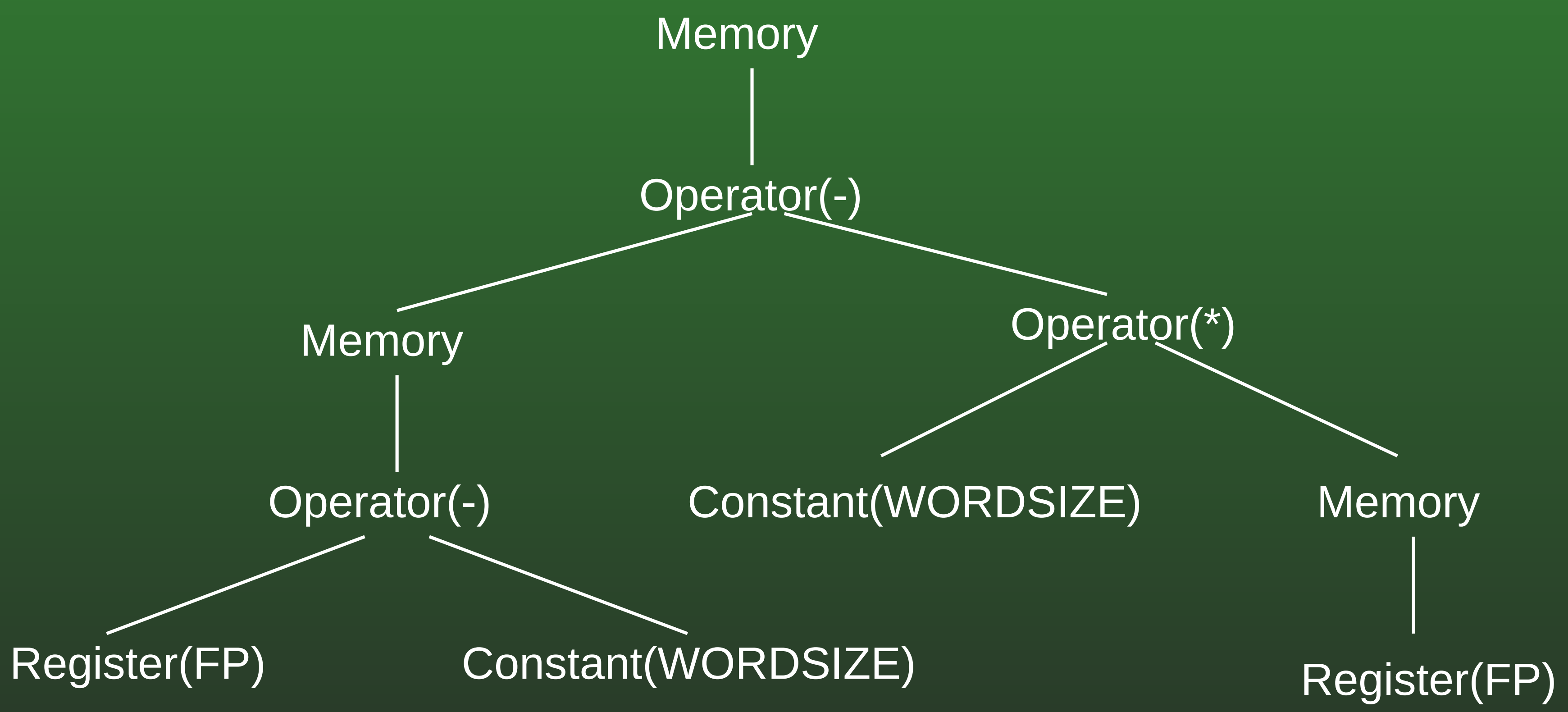


FR-123: Array Variables

- `A[x]`

FR-124: Array Variables

- `A[x]`

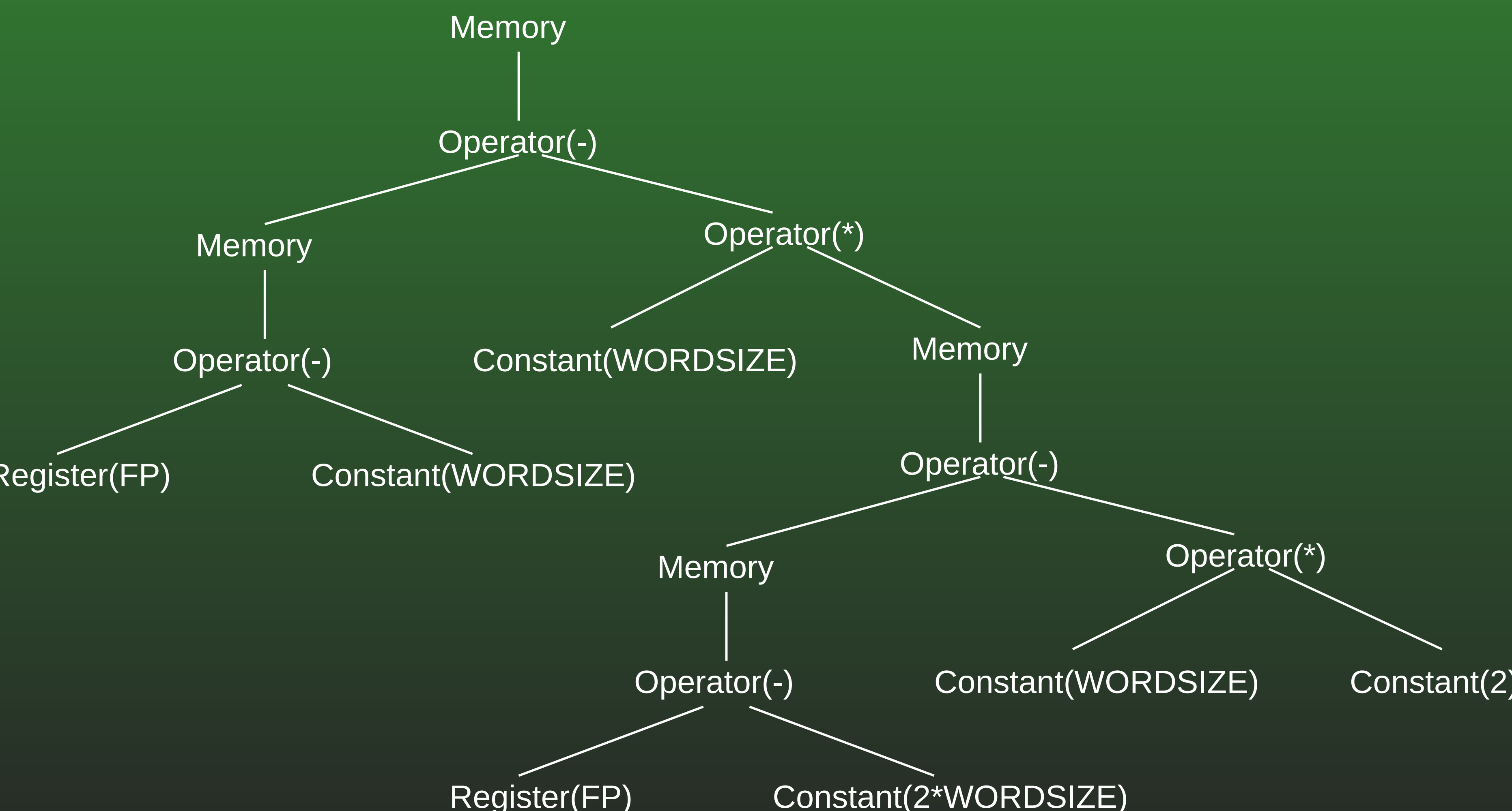


FR-125: Array Variables

- `A[B[2]]`

Array Variables

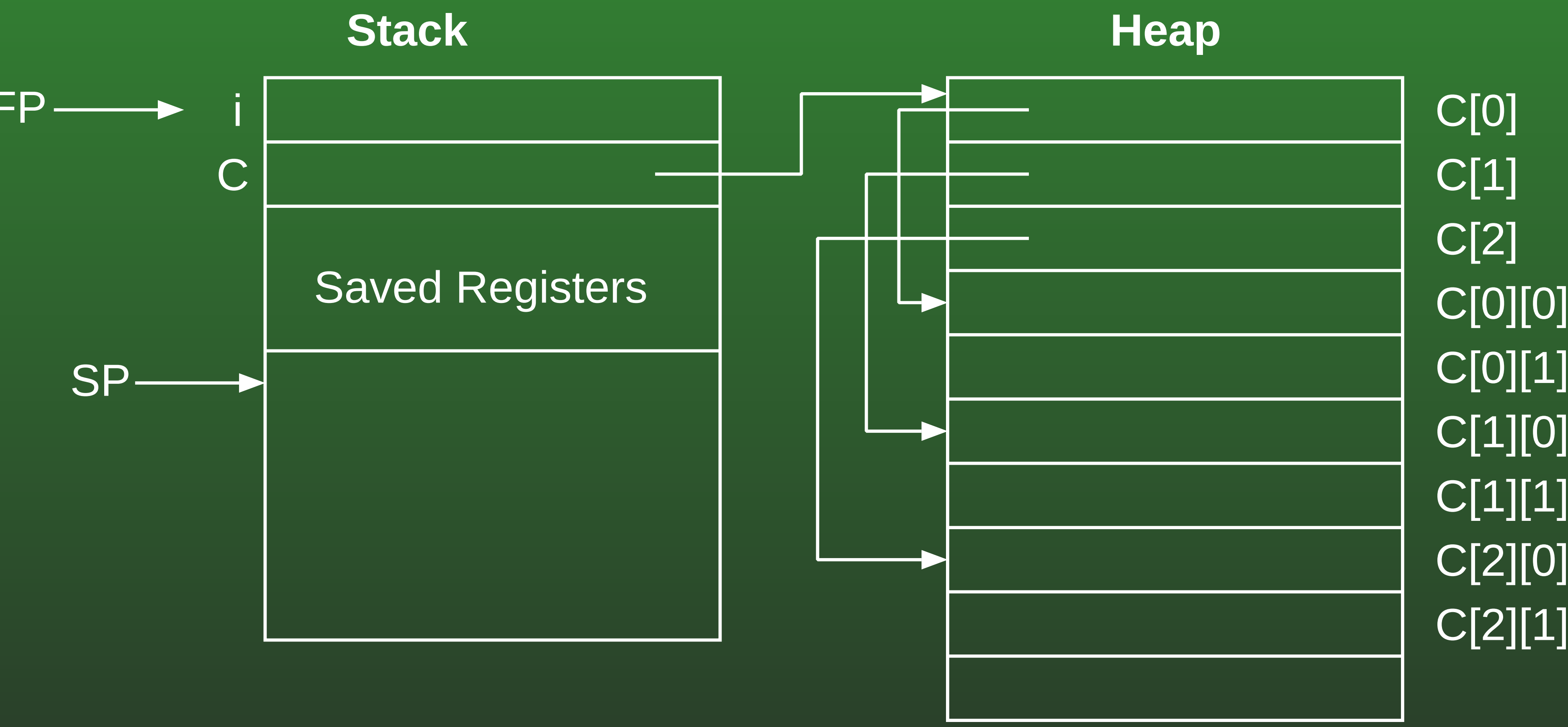
- `A[B[2]]`



FR-127: 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  */
}
```

FR-128: 2D Arrays

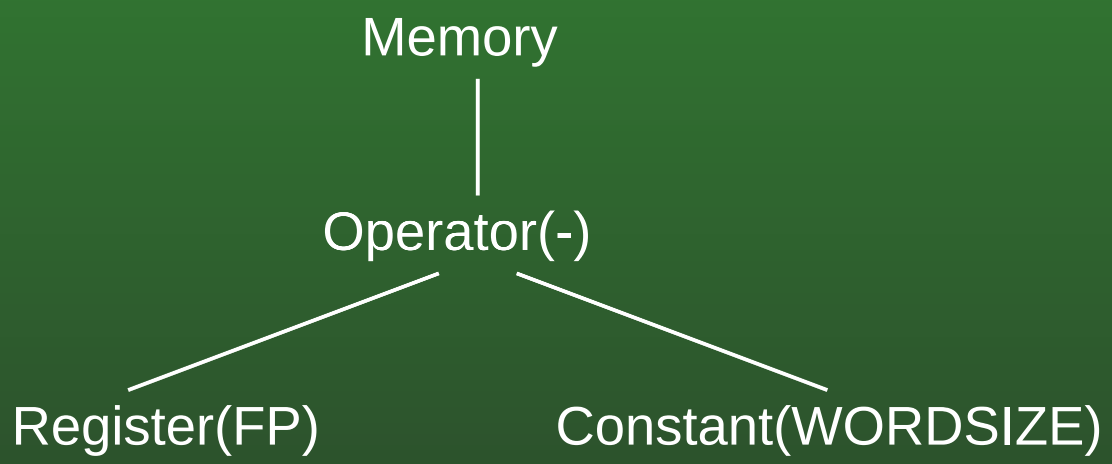


FR-129: 2D Arrays

- C

FR-130: 2D Arrays

- C

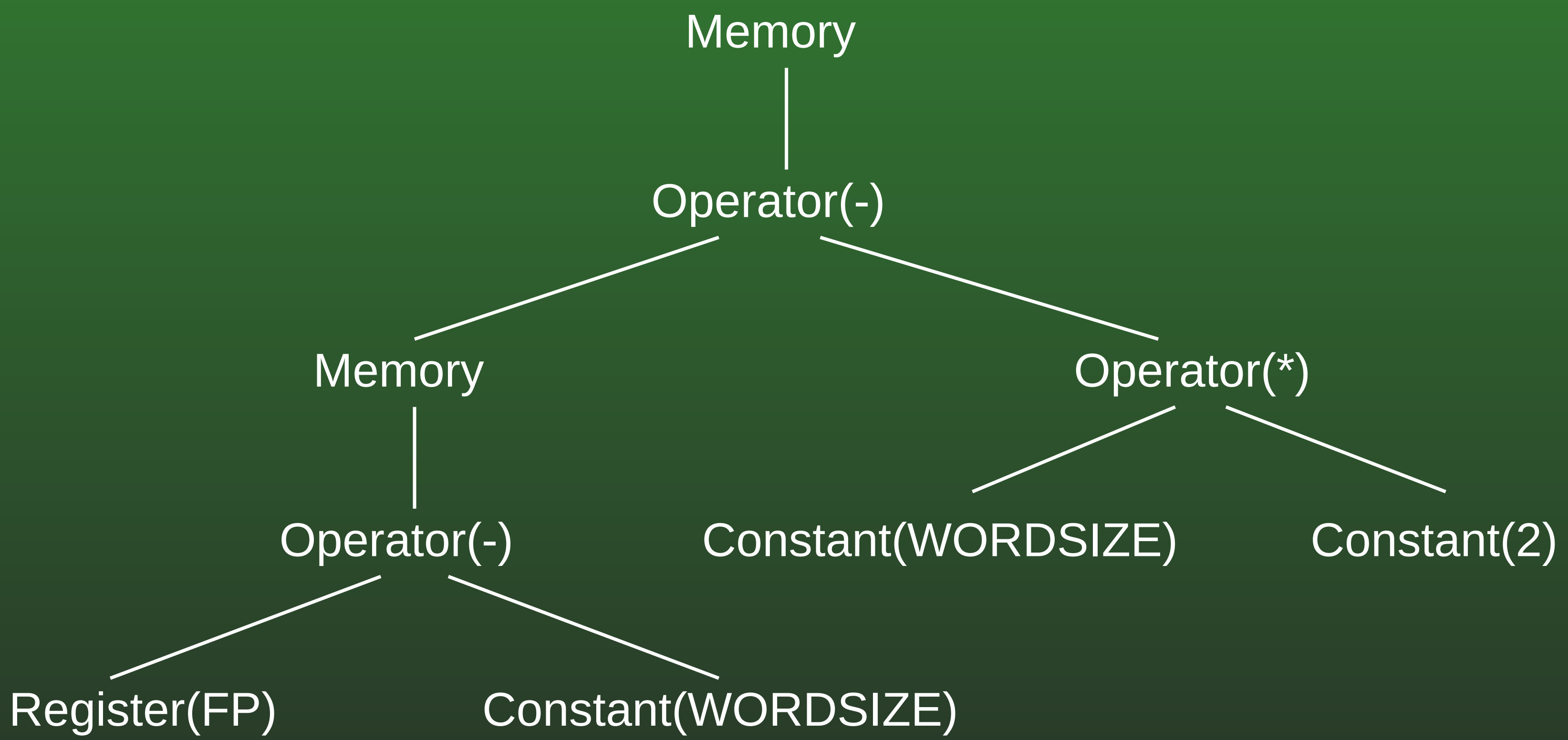


FR-131: 2D Arrays

- C[2]

FR-132: 2D Arrays

- C[2]

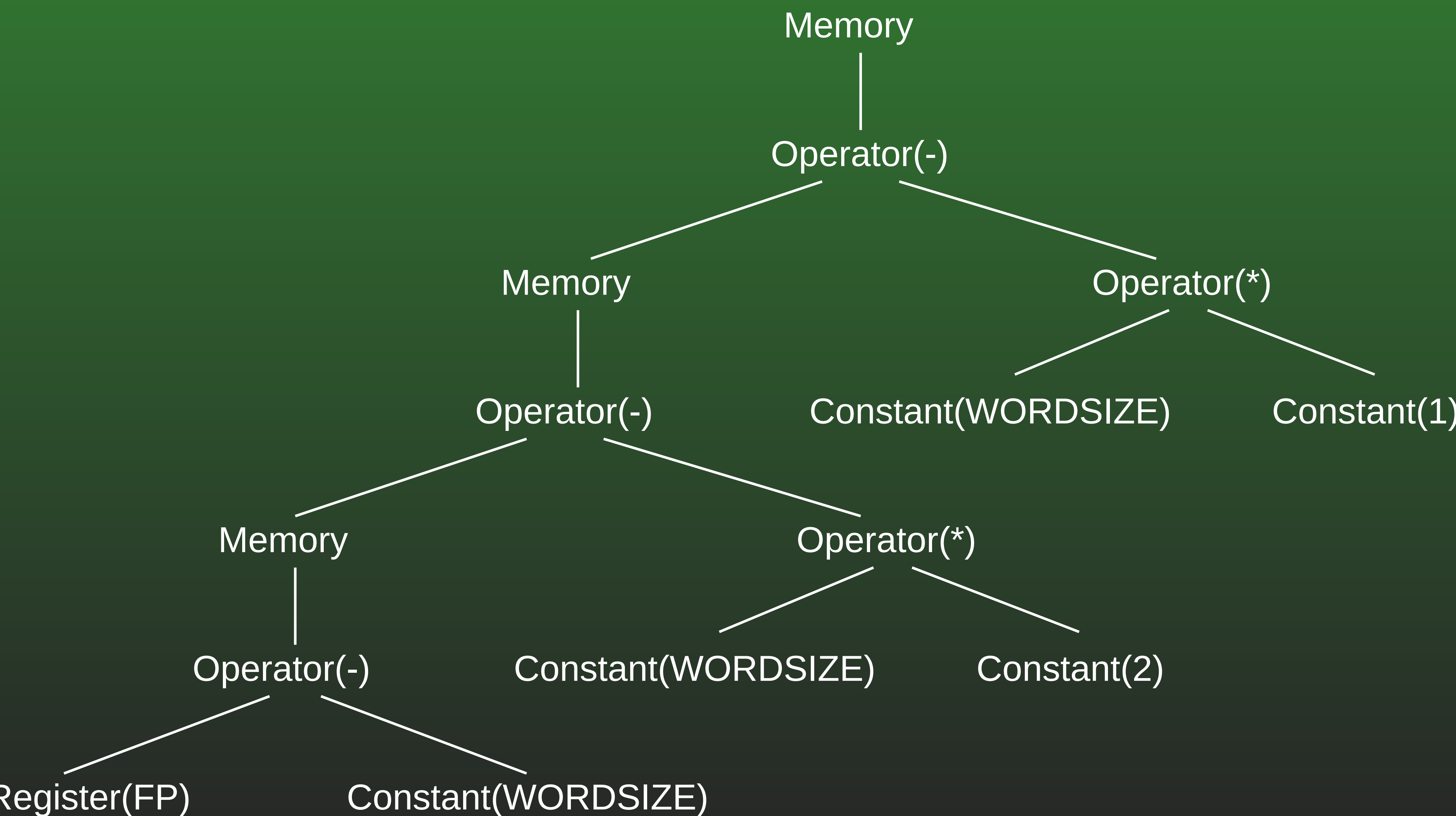


FR-133: 2D Arrays

- C[2][1]

FR-134: 2D Arrays

- C[2][1]



FR-135: 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

FR-136: 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 */
}
```

FR-137: Instance Variables

- Variable s

FR-138: Instance Variables

- Variable s

Memory

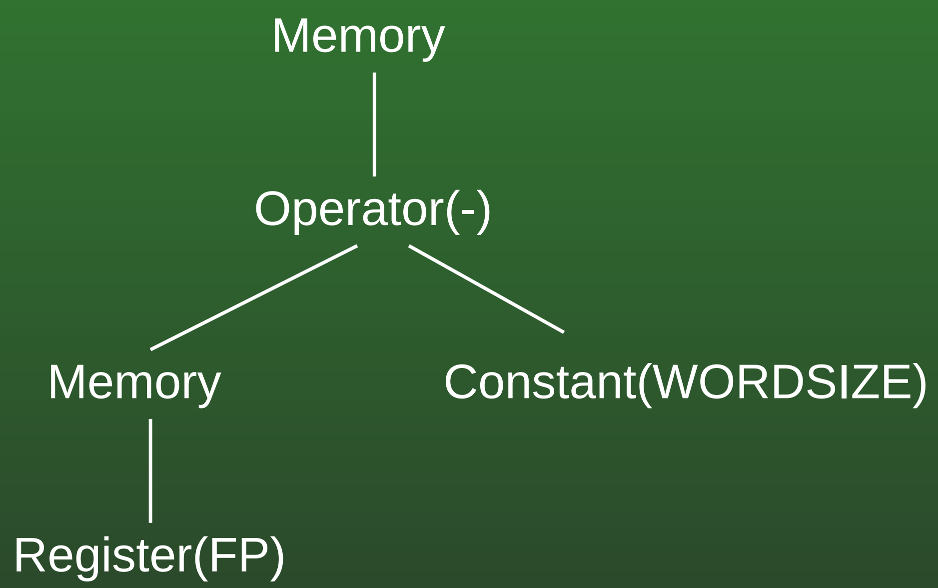
Register(FP)

FR-139: Instance Variables

- Variable `s.y`

FR-140: Instance Variables

- Variable `s.y`

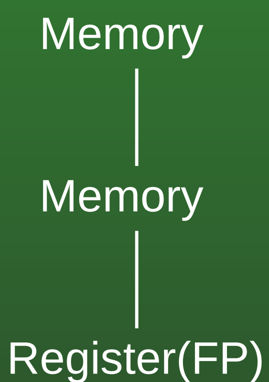


Instance Variables

- Variable `s.x`

FR-142: Instance Variables

- Variable `s.x`

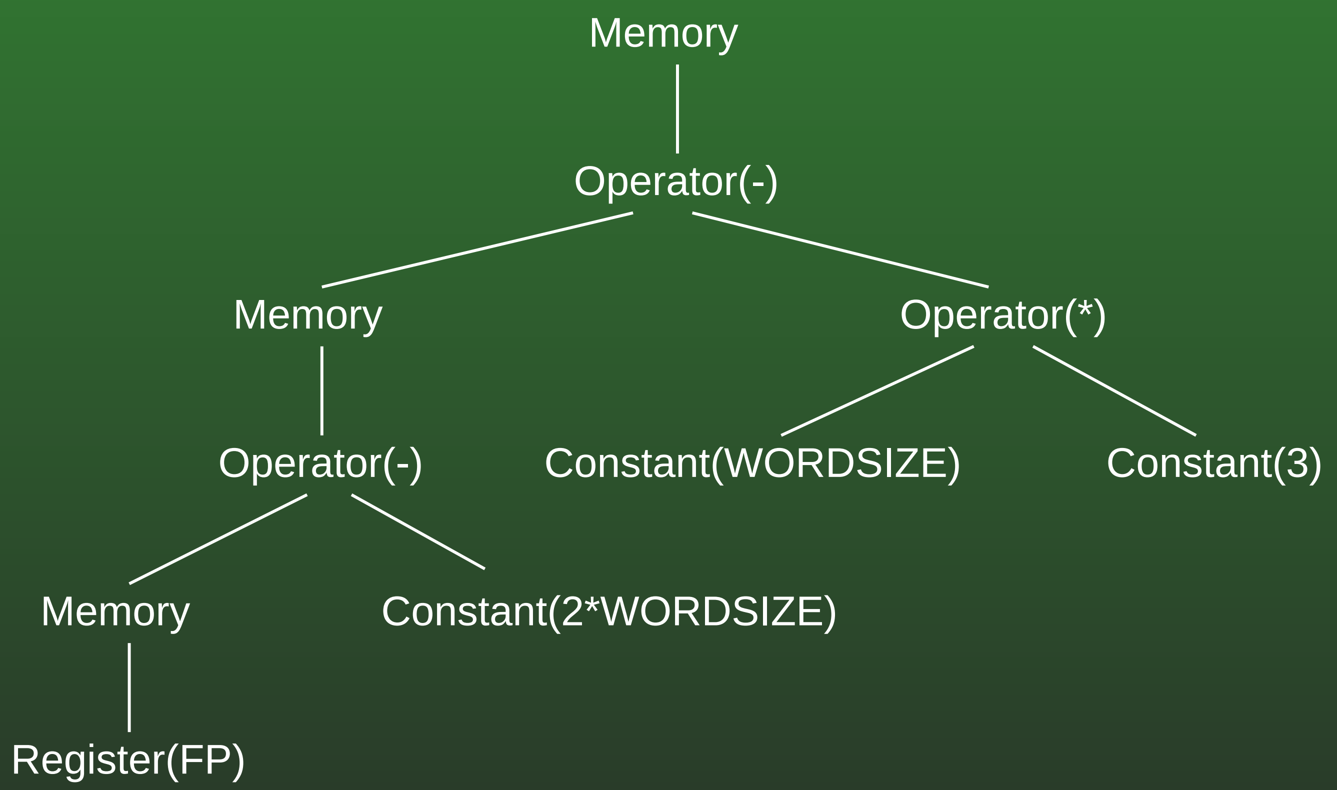


FR-143: Instance Variables

- Variable `s.A[3]`

FR-144: Instance Variables

- Variable `s.A[3]`

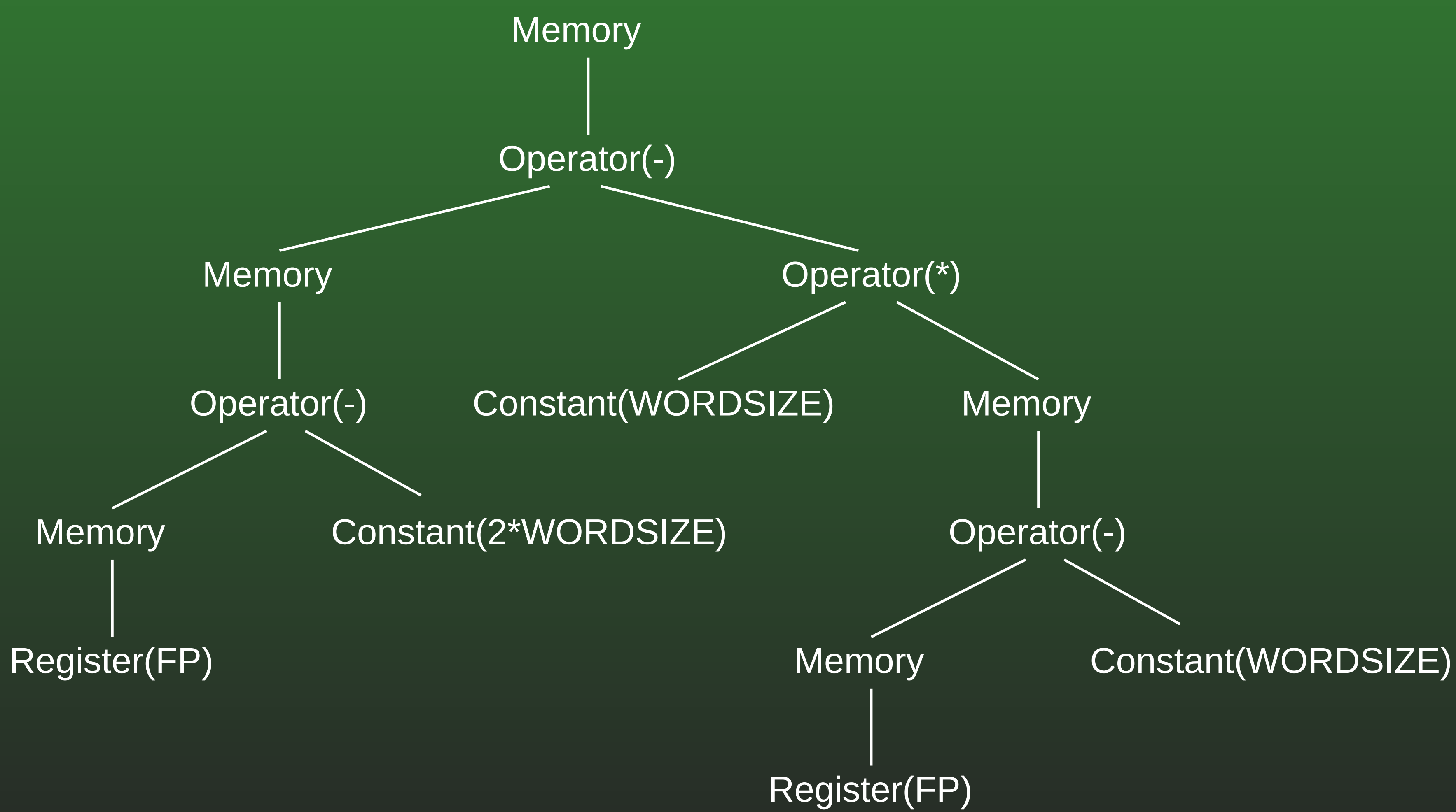


Instance Variables

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

Instance Variables

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



FR-147: While Statements

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

FR-148: While Statements

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

- Straightforward version:

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

FR-149: While Statements

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

- More Efficient:

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

Code Generation

- Next Step: Create actual assembly code.
- Use a tree tiling strategy:
  - Create a set of tiles with associated assembly code.
  - Cover the AST with these tiles.
  - Output the code associated with each tiles.
- As long as we are clever about the code associated with each tile, and how we tile the tree, we will create correct actual assembly.

FR-151: Tree Tilings

- We will explore several different tree tiling strategies:
  - Simple tiling, that is easy to understand but produces inefficient code
  - More complex tiling, that relies less on the expression stack
  - Modifications to complex tilings, to increase efficiency

FR-152: Simple Tiling

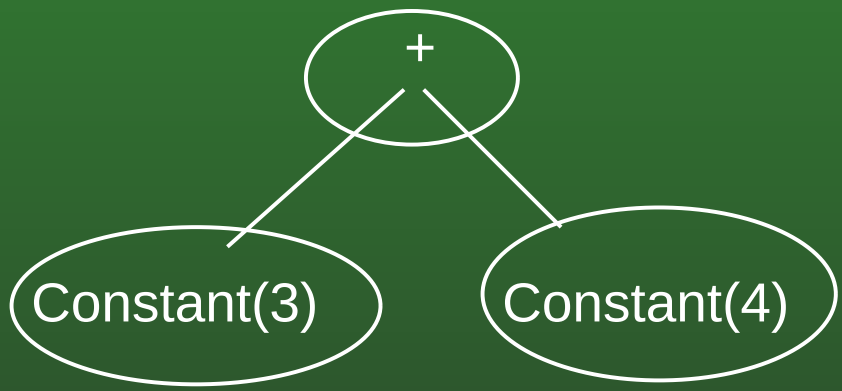
- Based on a post-order traversal of the tree
- Cover the tree with tiles
  - Each tile associated with actual assembly
- Emit the code associated with the tiles in a left-to-right, post-order traversal of the tree

FR-153: Simple Tiling

- Expression Trees
  - The code associated with an expression tree will place the value of that expression on the top of the expression stack

FR-154: Expression Trees

- Arithmetic Binary Operations



- What should the code for the + tile be?
  - Code for entire tree needs to push *just* the final sum on the stack
  - Code for Constant Expressions push constant values on top of the stack

FR-155: Expression Trees

- Arithmetic Binary Operations
  - Code for a “+” tile:

```
lw   $t1, 8($ESP)     % load first operand
lw   $t2, 4($ESP)     % load the second operand
add  $t1, $t1, $t2    % do the addition
sw   $t1, 8($ESP)     % store the result
add  $ESP, $ESP, 4    % update the ESP
```

FR-156: Expression Trees

- Memory Accesses
  - Memory node is a memory dereference
    - Pop operand into a register
    - Dereference register
    - Push result back on stack

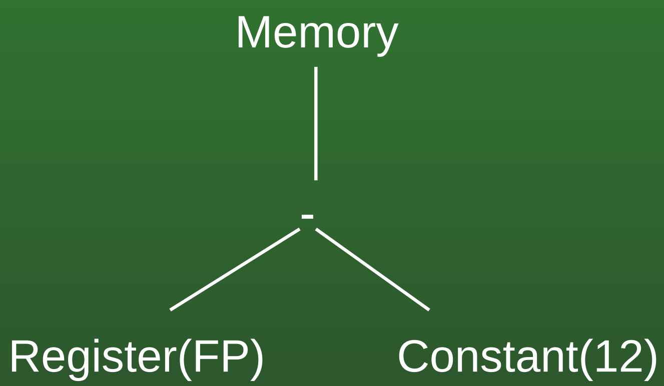
Expression Trees

- Memory Accesses

```
lw $t1, 4($ESP)
lw $t1, 0($t1)
sw $t1, 4($ESP)
```

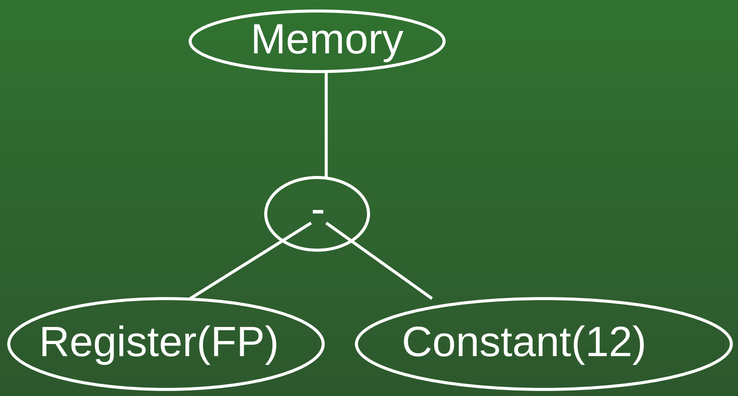
Expression Trees

- Memory Example

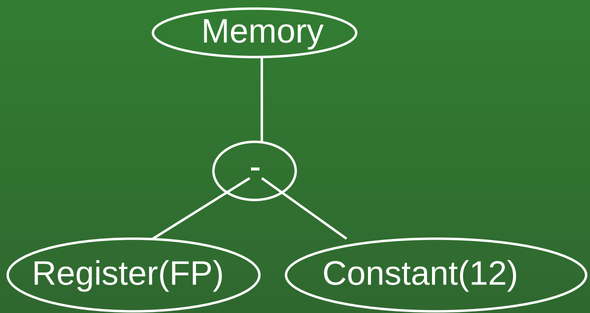


FR-159: Expression Trees

- Memory Example



FR-160: Expression Trees



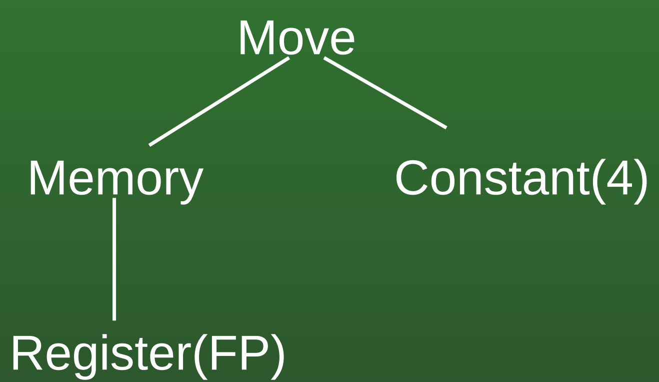

```
sw   $FP,  0($ESP)     % Store frame pointer on the top of the expressin stack
addi $ESP, $ESP,  -4   % Update the expression stack pointer
addi $t1,  $zero, 12   % Load the constant value 12 into the register $t1
sw   $t1,  0($ESP)     % Store $t1 on the top of the expression stack
addi $ESP, $ESP,  -4   % update the expression stack pointer
lw   $t1,  4($ESP)     % load the first operand into temporary $t1
lw   $t2,  8($ESP)     % load the second operand into temparary $t2
sub  $t1,  $t1,    $t2 % do the subtraction, storing result in $t1
sw   $t1,  8($ESP)     % store the result on the expression stack
add  $ESP, $ESP,   4   % update the expression stack pointer
lw   $t1,  4($ESP)     % Pop the address to dereference off the top of
                       % the expression stack
lw   $t1, 0($t1)       % Dereference the pointer
sw   $t1, 4($ESP)      % Push the result back on the expression stack
```

FR-161: **Simple Tiling**

- Statement Trees
  - The code associated with a statement tree implements the statement described by the tree

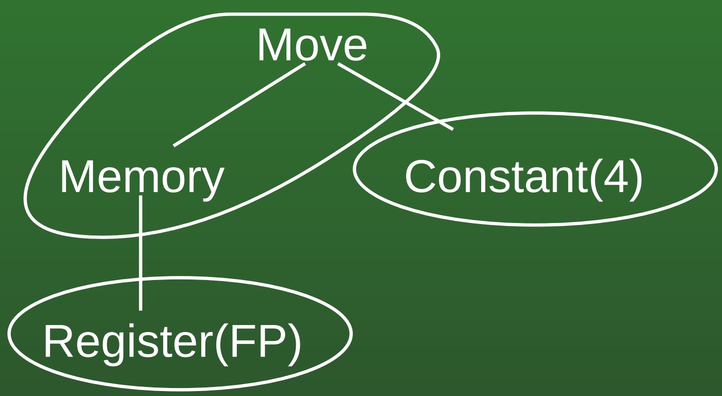
FR-162: Statement Trees

- Move Trees (Moving into MEMORY locations)



Statement Trees

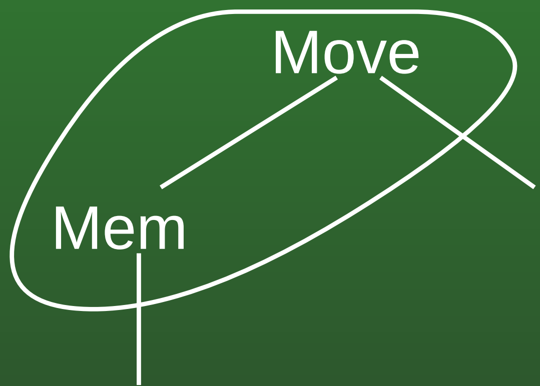
- Move Trees (Moving into MEMORY locations)



- The code for the MOVE tile needs to:
  - Pop the value to move off the stack
  - Pop the destination of the move off the stack
  - Store the value in the destination

Statement Trees

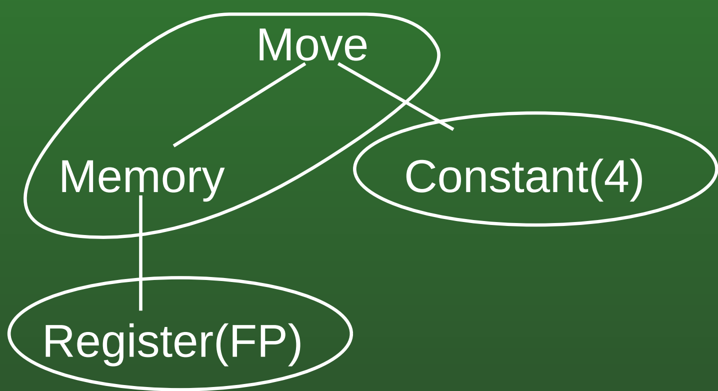
- Move Trees (Moving into MEMORY locations)




```
lw   $t1,  8($ESP)
lw   $t2,  4($ESP)
sw   $t2,  0($t1)
addi $ESP, $ESP, 8
```

FR-165: Statement Trees

- Move Trees (Moving into MEMORY locations)




```
sw   $FP,  0($ESP)    % Store the frame pointer on the expression stack
addi $ESP, $ESP, -4   % Update the expression stack pointer
addi $t1,  $ZERO, 4   % Put constant 4 into a register
sw   $t1,  0($ESP)    % Store resgister on the expression stack
addi $ESP, $ESP, -4   % Update expression stack pointer
lw   $t1,  8($ESP)    % Store the address of the lhs of the move in a regiter
lw   $t2,  4($ESP)    % Store value of the rhs of the move in a register
sw   $t2,  0($t1)     % Implement the move
addi $ESP, $ESP, 8    % update the expression stack pointer
```

FR-166: Improved Tiling

- Tiling we've seen so far is correct – but inefficient
  - Generated code is much longer than it needs to be
  - Too heavy a reliance on the stack (main memory accesses are slow)
- We can improve our tiling in three ways:

FR-167: Improved Tiling

- Decrease reliance on the expression stack
- Use large tiles
- Better management of the expression stack
  - Including storing the bottom of the expression stack in registers

FR-168: Accumulator Register

- Code for expression trees will no longer place the value on the top of the expression stack
- Instead, code for expression trees will place the value of the expression in an accumulator register (ACC)
- Stack will still be necessary (in some cases) to store partial values

Accumulator Register

- Constant trees
    - Tree: Constant(15)
    - Code:

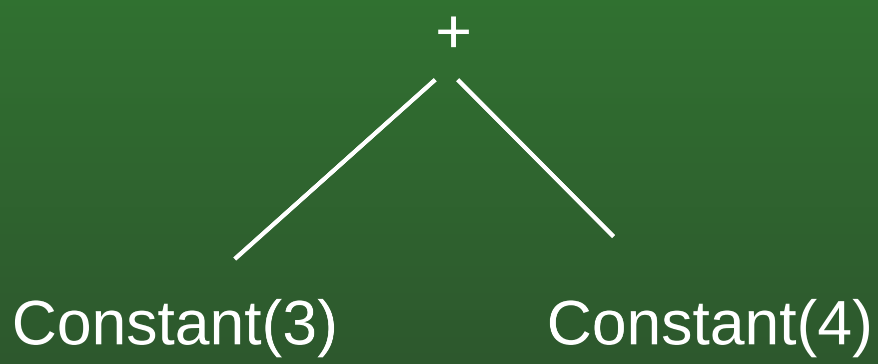
```
addi $ACC, $zero, 15
```

FR-170: Accumulator Register

- Binary Operators (+, -, *, etc)
  - Use an INORDER traversal instead
    - Emit code for left subtree
    - Store this value on the stack
    - Emit code for the right subtree
    - Pop value of left operand off stack
    - Do the operation, storing result in ACC

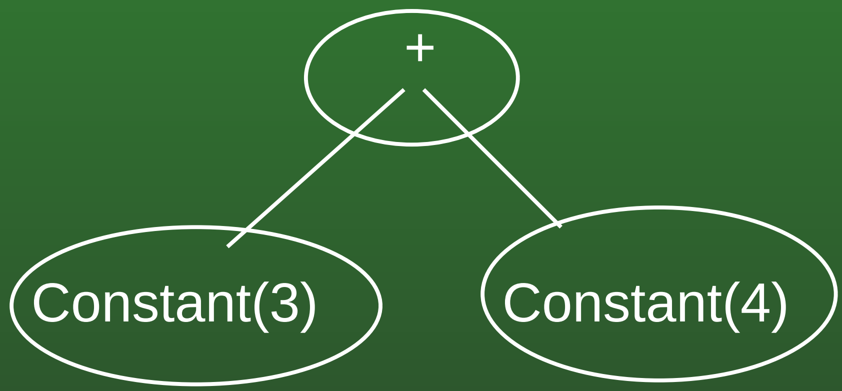
FR-171: Accumulator Register

- Binary Operators (+, -, *, etc)



FR-172: Accumulator Register

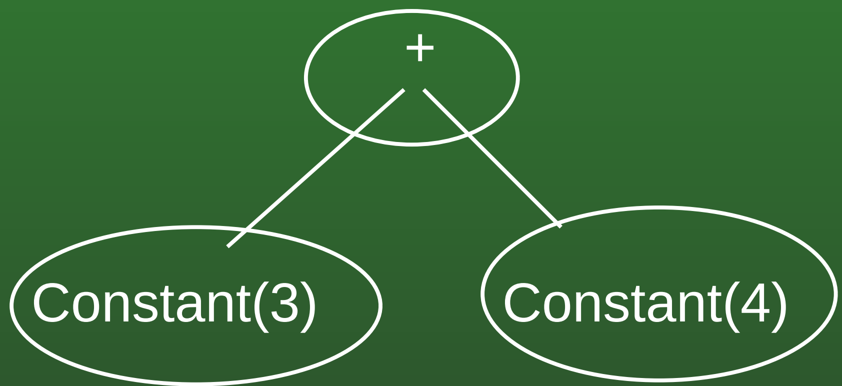
- Binary Operators (+, -, *, etc)



- Emit code for left subtree
- Push value on stack
- Emit code for right subtree
- Do arithmetic, storing result in ACC

FR-173: Accumulator Register

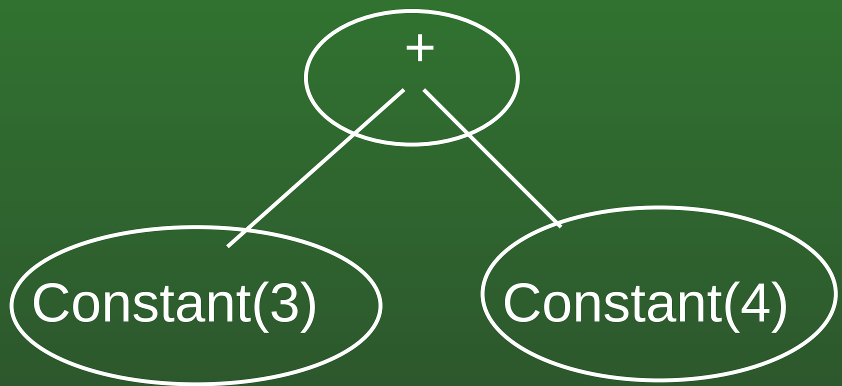
- Binary Operators (+, -, *, etc)




```
<code for left operand>
sw   $ACC, 0($ESP)
addi $ESP, $ESP, -4
<code for right operand>
lw   $t1,  4($ESP)
addi $ESP, $ESP, 4
add  $ACC, $t1, $ACC
```

FR-174: Accumulator Register

- Binary Operators (+, -, *, etc)



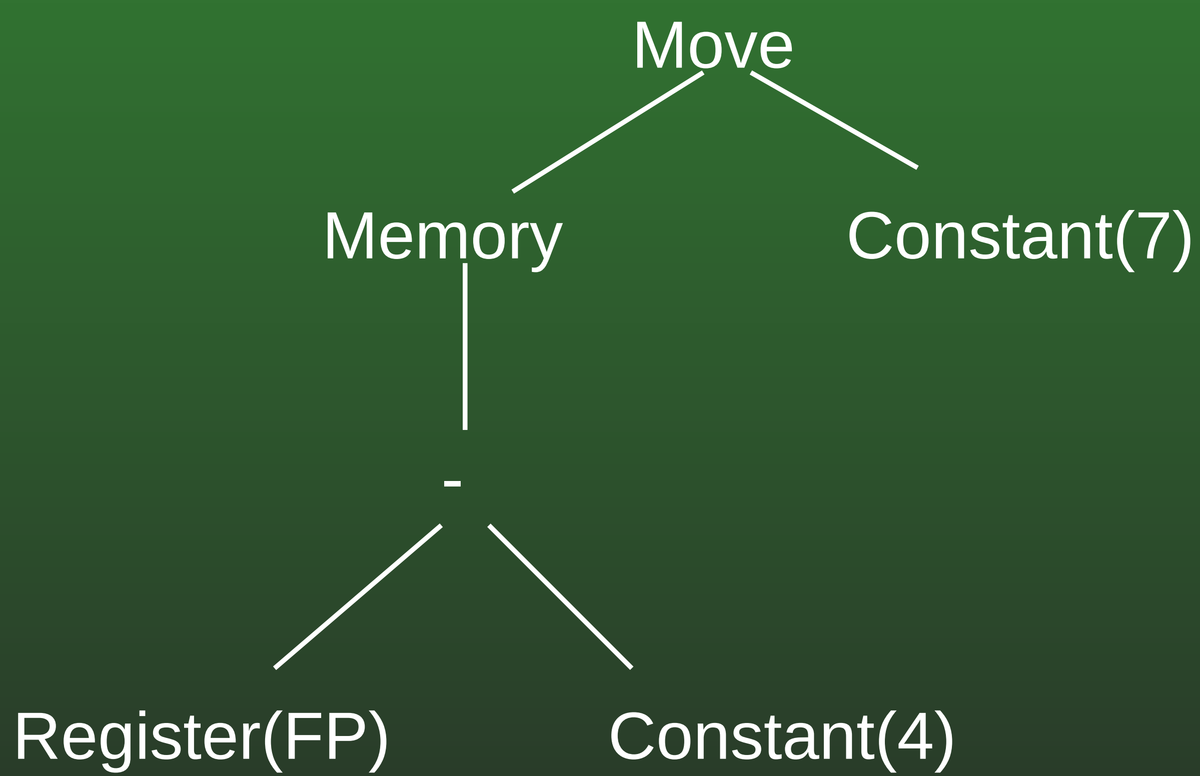

```
addi $ACC, $zero, 3
sw   $ACC, 0($ESP)
addi $ESP, $ESP, -4
addi $ACC, $zero, 4
lw   $t1,  4($ESP)
addi $ESP, $ESP, 4
add  $ACC, $t1, $ACC
```

FR-175: Larger Tiles

- Instead of covering a single node for each tile, cover several nodes with the same tile
- As long as the code associated with the larger tile is more efficient than the code associated with all of the smaller tiles, we gain efficiency

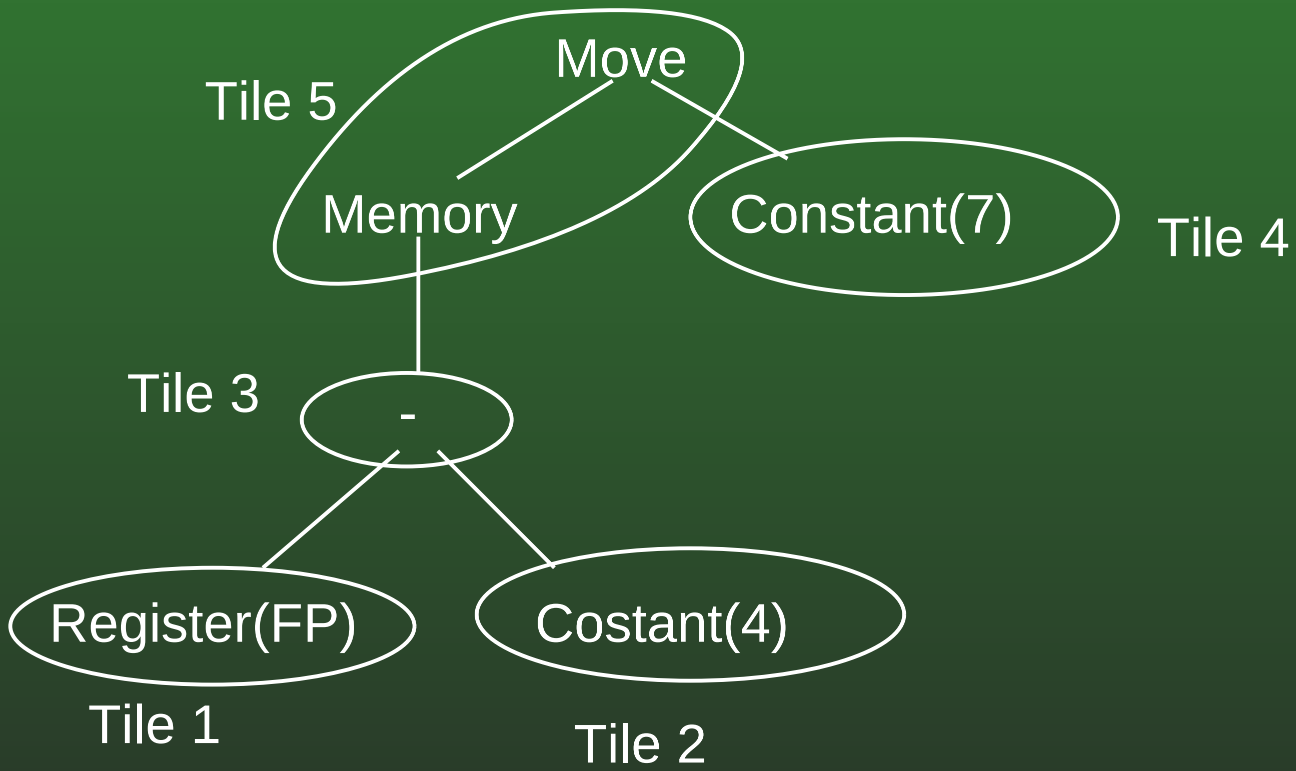
Larger Tiles Example

- Memory Move Expression



Larger Tiles Example

- Standard Tiling



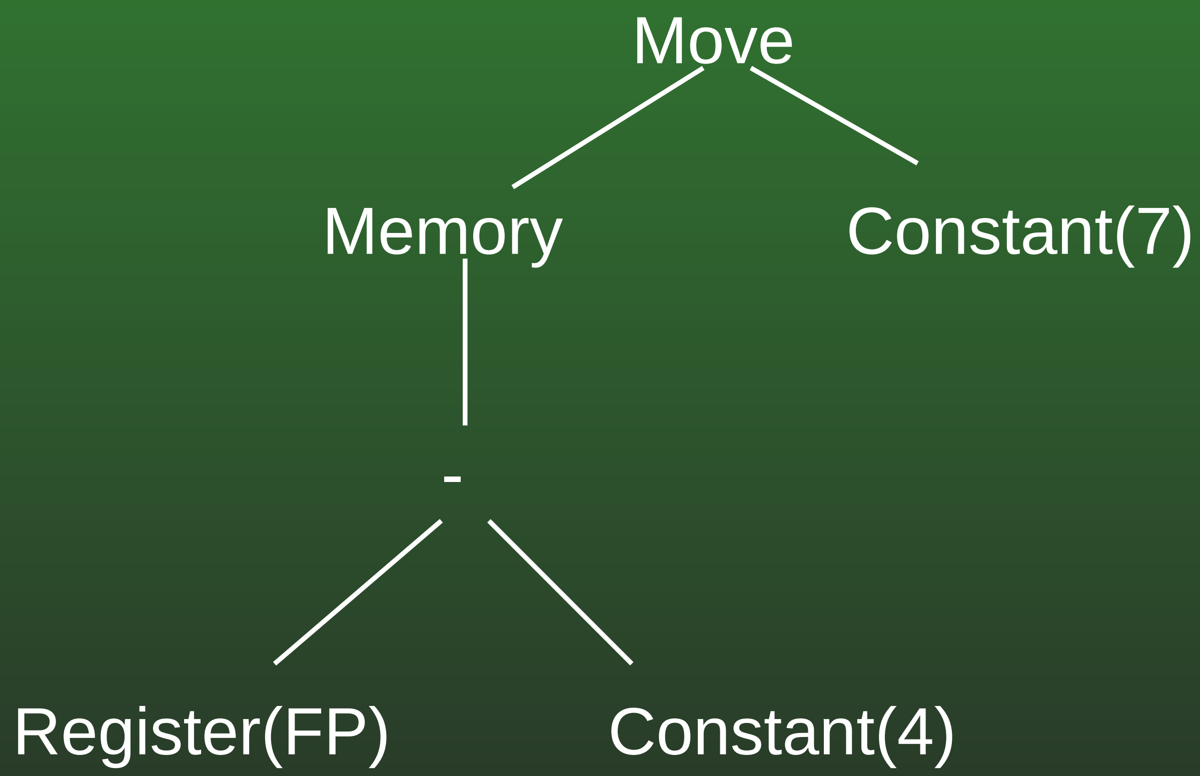
FR-178: Larger Tiles Example

- Standard Tiling

```
addi  $ACC, $FP, 0      % code for tile 1
sw    $ACC, $ESP, 0     % code for tile 3
addi  $ESP, $ESP, -4    % code for tile 3
addi  $ACC, $zero, 4    % code for tile 2
lw    $t1,  4($ESP)     % code for tile 3
addi  $ESP, $ESP, 4     % code for tile 3
sub   $ACC, $t1, $ACC   % code for tile 3
sw    $ACC, $ESP, 0     % code for tile 5
addi  $ESP, $ESP, -4    % code for tile 5
addi  $ACC, $zero, 7    % code for tile 4
lw    $t1,  4($ESP)     % code for tile 5
addi  $ESP, $ESP,4      % code for tile 5
sw    $ACC, 0($t1)      % code for tile 5
```

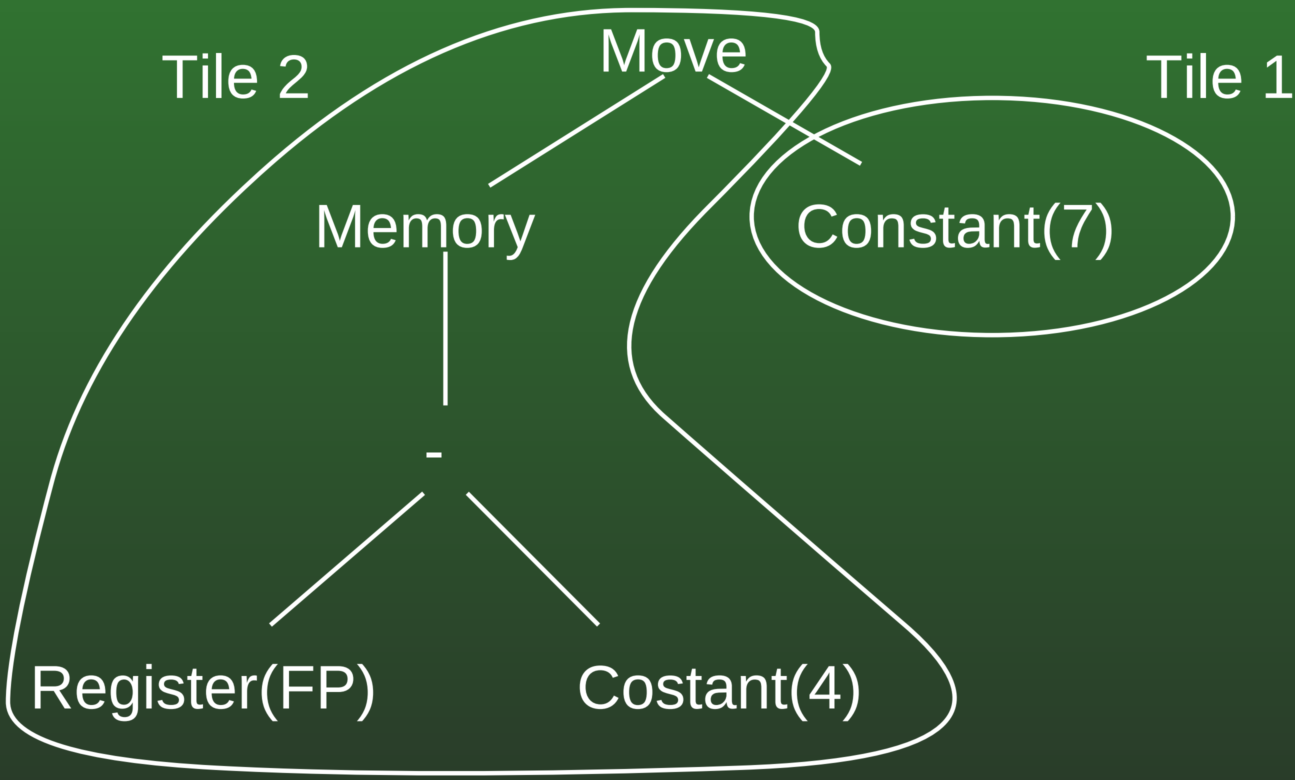
FR-179: Larger Tiles Example

- Memory Move Expression

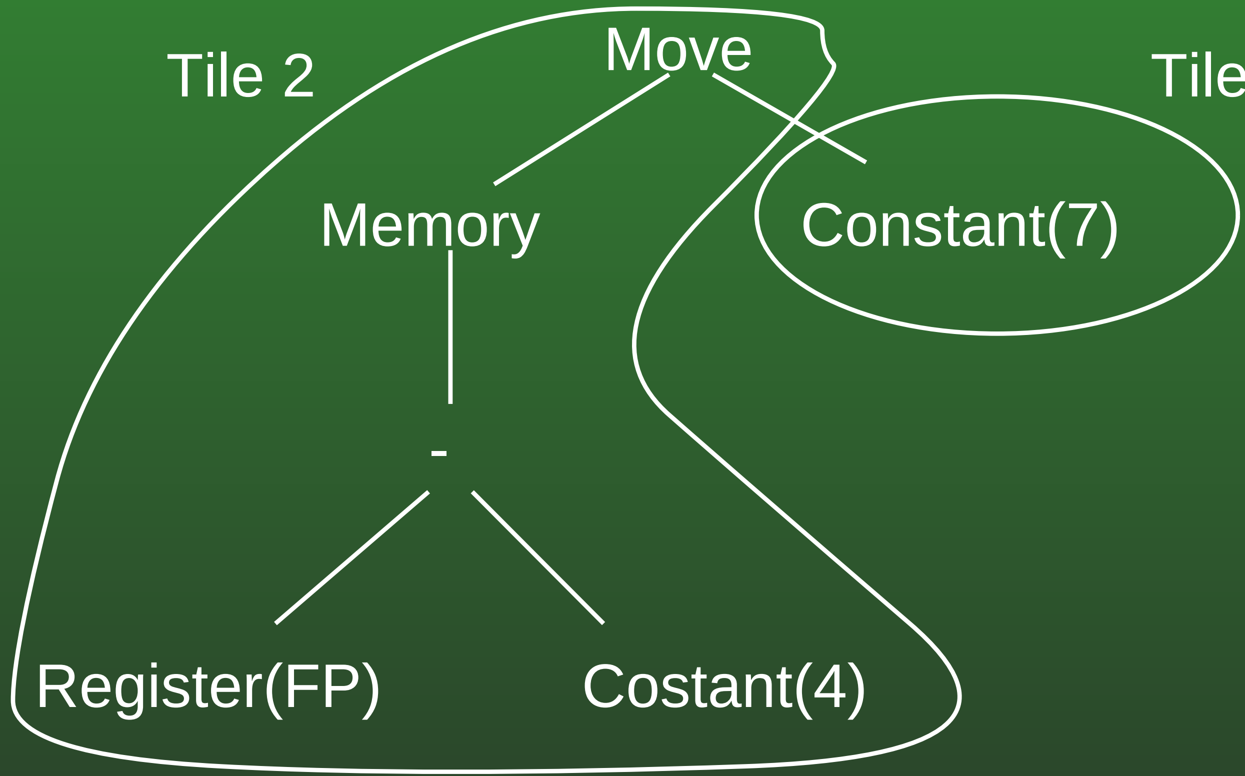


FR-180: Larger Tiles Example

- Using Larger Tiles

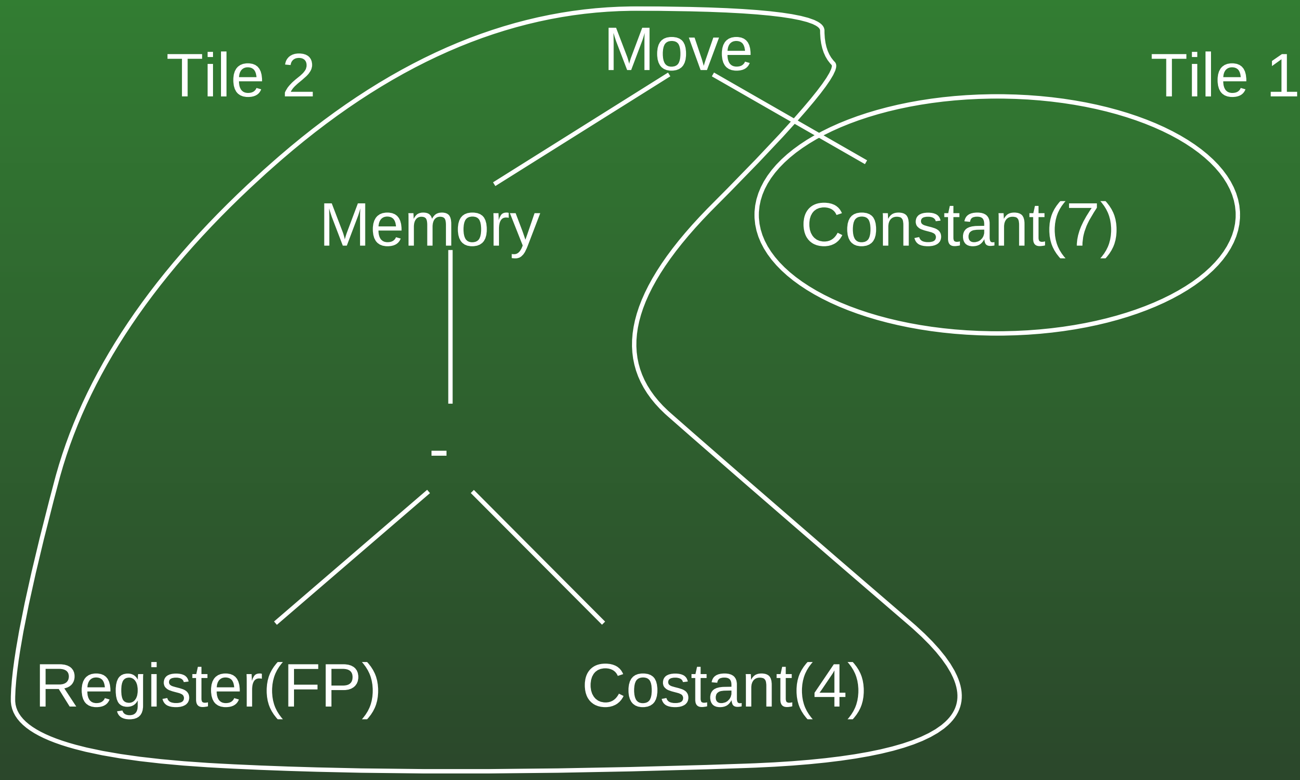


Larger Tiles Example



- Code for Tile 2?

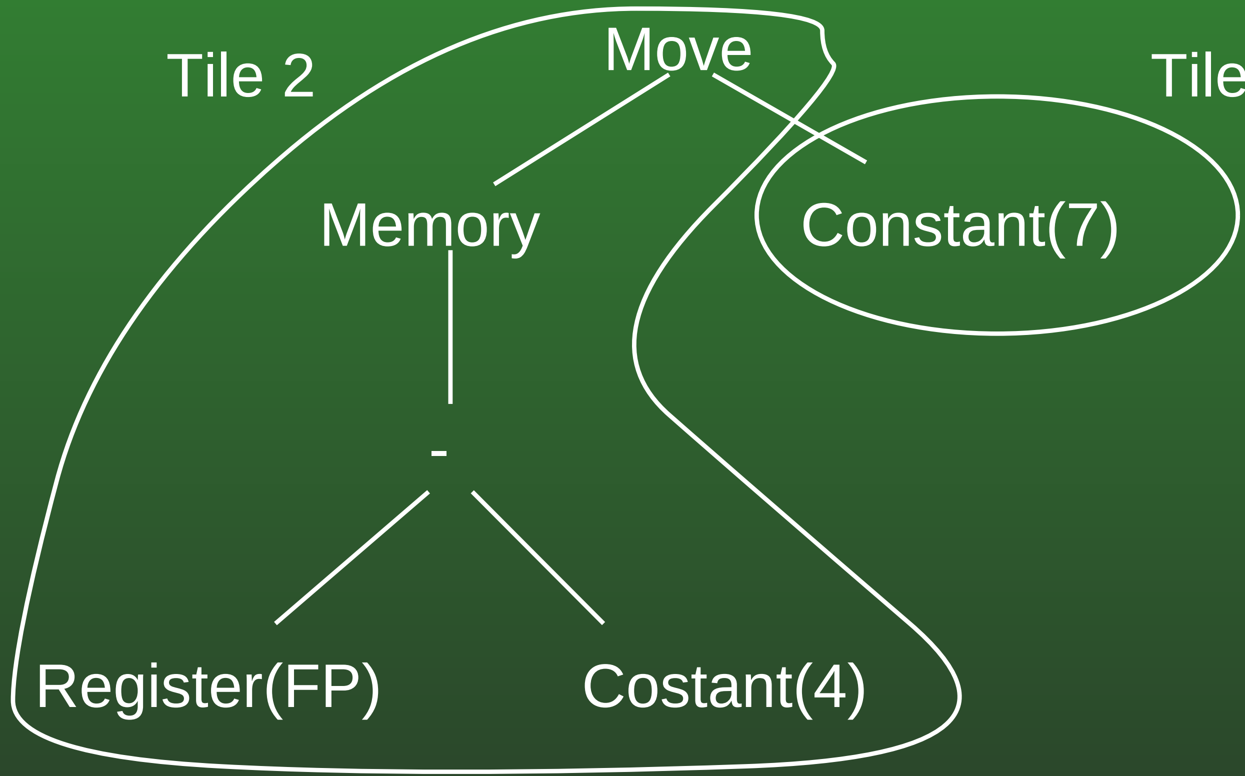
Larger Tiles Example



- Code for Tile 2?

```
sw  $ACC, -4($FP)
```

FR-183: Larger Tiles Example




```
addi   $ACC, $zero 7       % tile 1
sw     $ACC, -4($FP)       % tile 2
```

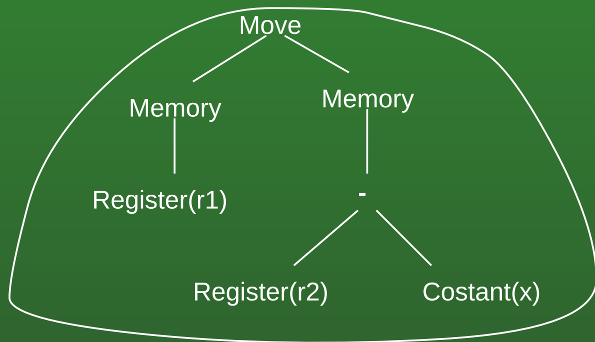
Adding New Instructions

- Added a new assembly language instruction:

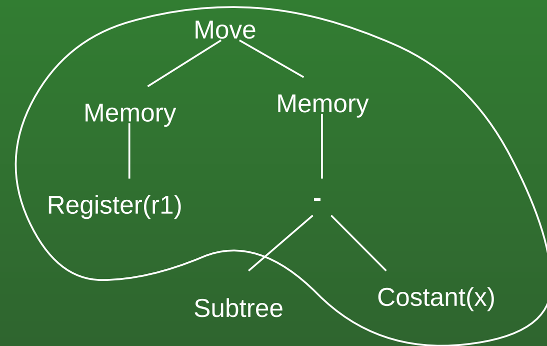
```
loadmem $r1, <offset>($r2)
```

- MEMORY[$r1] = MEMORY[$r2 + offset]
- Create a tileset for this new instruction

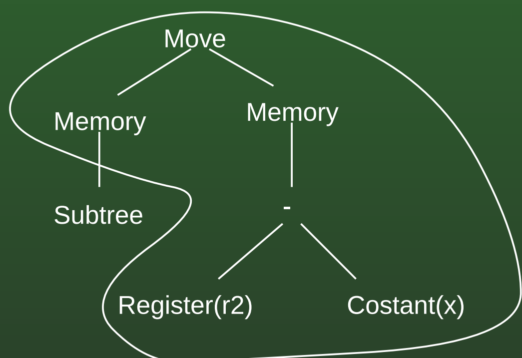
FR-185: Adding New Instructions



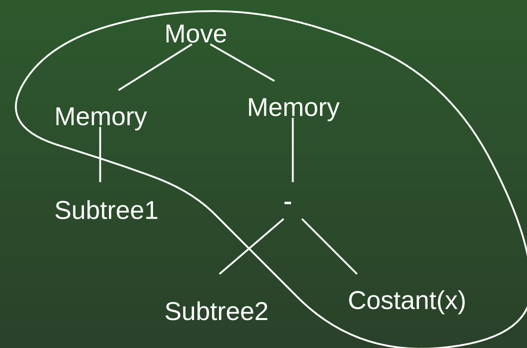

```
loadmem r1, -x(r2)
```




```
<code for Subtree>
loadmem $r1, -x($ACC)
```

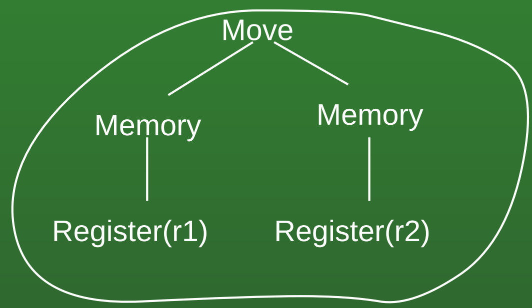



```
<code for Subtree>
loadmem $ACC, -x($r2)
```

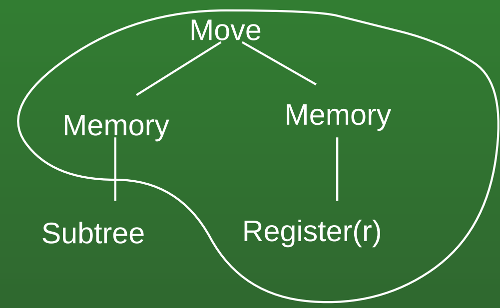



```
<code for Subtree1>
sw   $acc, 0($ESP)
addi $ESP, $ESP, -4
<code for Subtree2>
lw   $t1, 4($ESP)
addi $ESP, $ESP, 4
loadmem  $t1, -x($ACC)
```

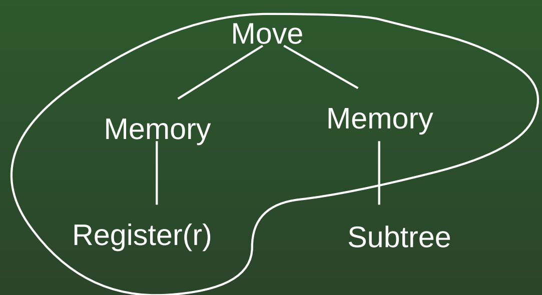
FR-186: Adding New Instructions



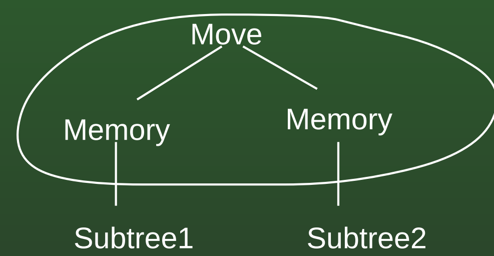

```
loadmem r1, 0($r2)
```




```
<code for Subtree>
loadmem $ACC, 0($r)
```

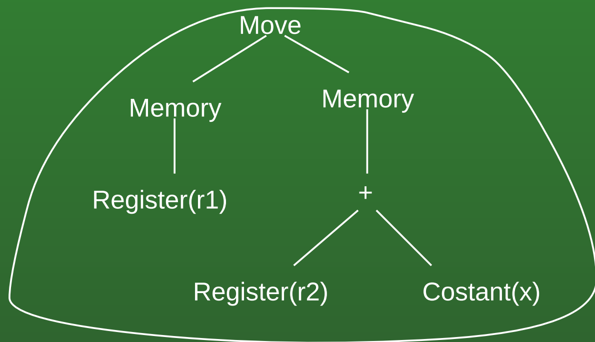



```
<code for Subtree>
loadmem $r, 0($ACC)
```

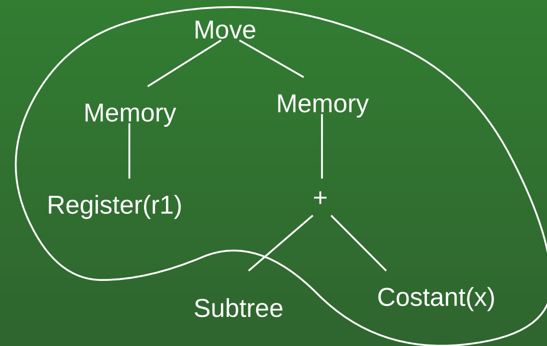



```
<code for Subtree1>
sw   $ACC, 0($ESP)
addi $ESP, $ESP, -4
<code for Subtree2>
lw   $t1, 4($ESP)
addi $ESP, $ESP, 4
loadmem  $t1, 0($ACC)
```

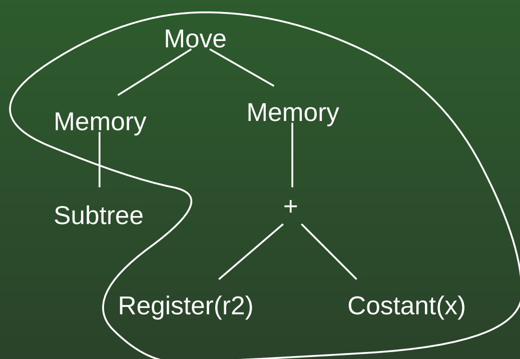
FR-187: Adding New Instructions



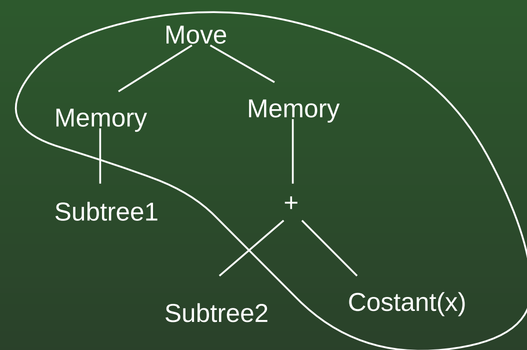

```
loadmem r1, x(r2)
```




```
<code for Subtree>
loadmem $r1, x($ACC)
```



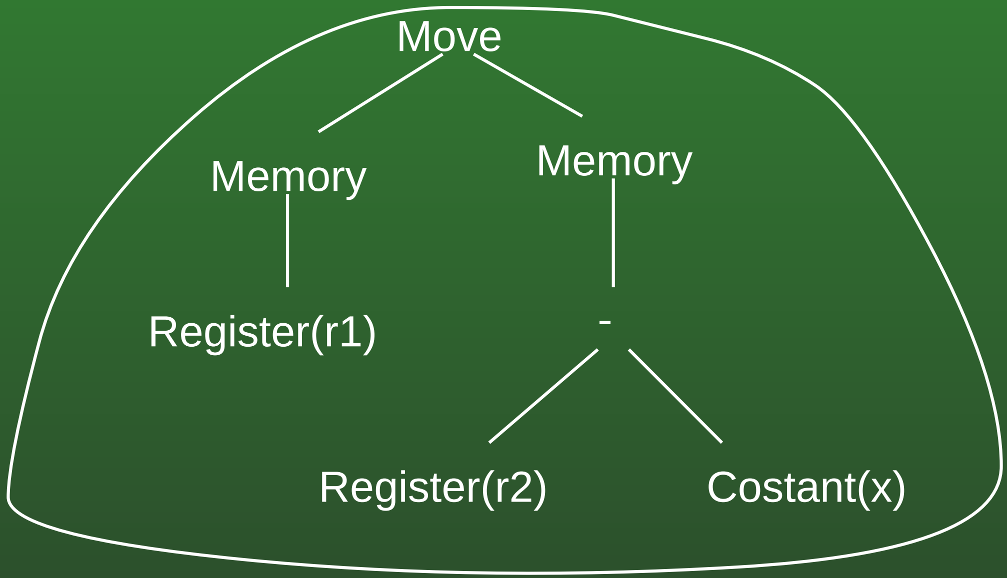

```
<code for Subtree>
loadmem $ACC, x($r2)
```




```
<code for Subtree1>
sw   $ACC, 0($ESP)
addi $ESP, $ESP, -4
<code for Subtree2>
lw   $t1, 4($ESP)
addi $ESP, $ESP, 4
loadmem  $t1, x($ACC)
```

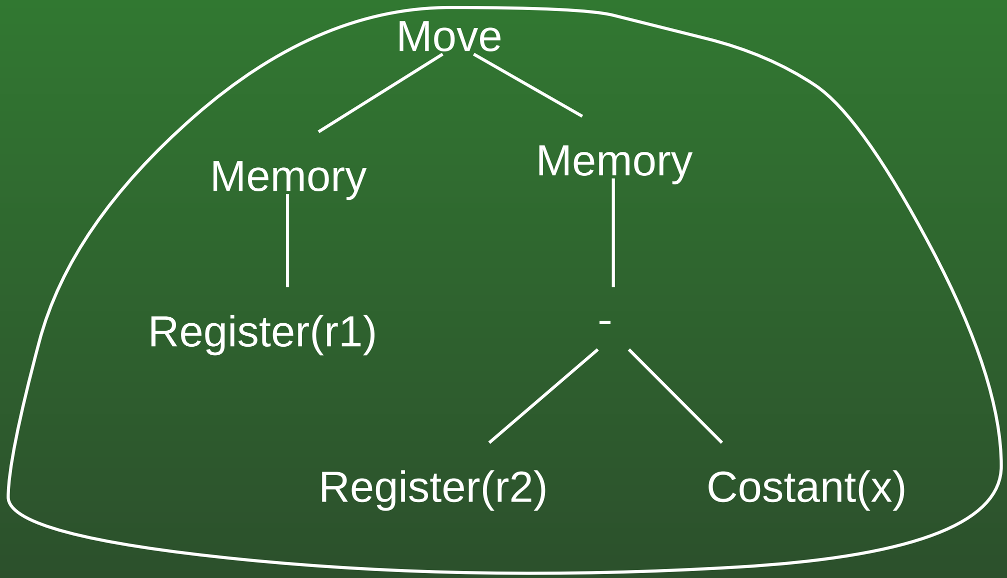

Adding New Instructions

- Code that uses this tile:



Adding New Instructions

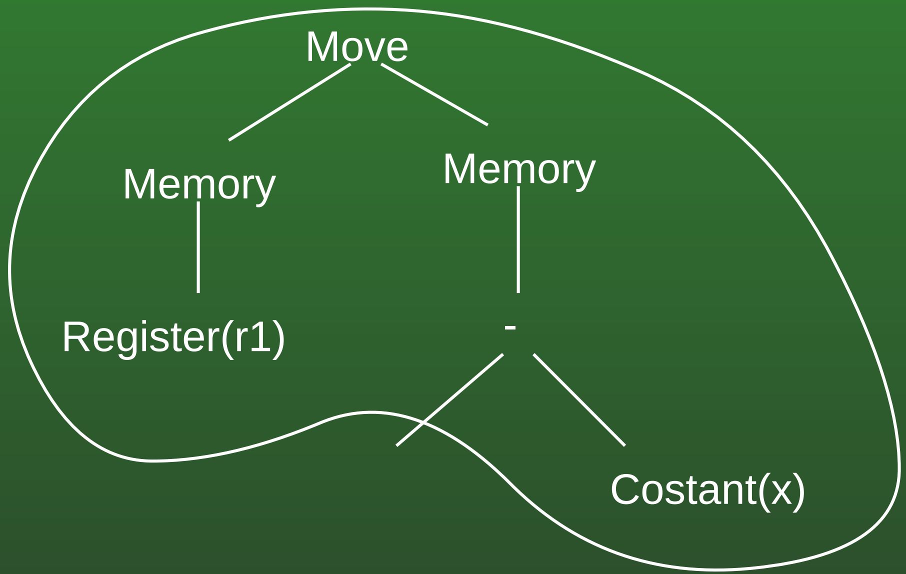
- Code that uses this tile:



- `x = y;`
  - if `x` is at offset 0

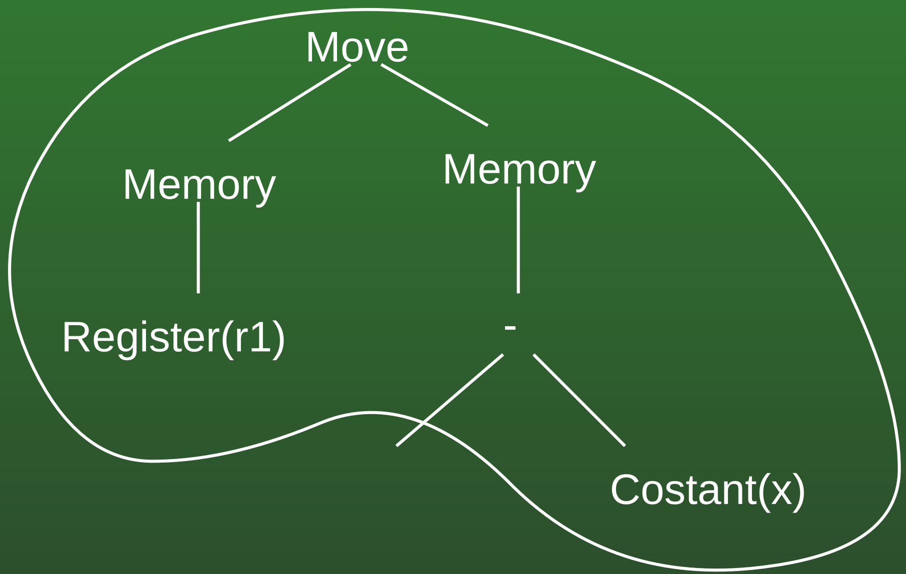
Adding New Instructions

- Code that uses this tile:



FR-191: Adding New Instructions

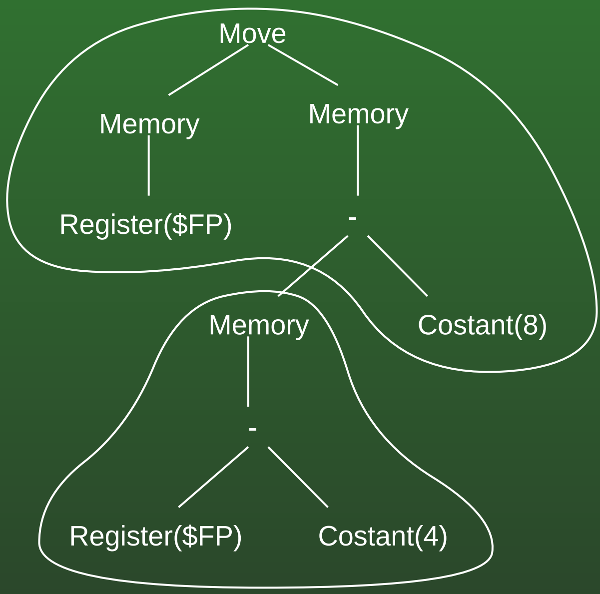
- simpleJava Code that uses this tile:



- `x = y.z;`
    - `x` is at offset 0

FR-192: Adding New Instructions

- `x = y.z;`
  - `x` is at offset 0, `y` is at offset 4, `z` is at offset 8




```
lw      $ACC, -8 ($FP)
loadmem $FP,  -4($ACC)
```

Questions on Final

You can expect to find the following types of questions on the final:

- Create a regular expression for a specifica token type
  - Floating point numbers, different bases, etc
- Create a CFG for a language fragment
- Give the parse tree several strings for a given CFG
- Show that a CFG is ambiguous

FR-194: Questions on Final

You can expect to find the following types of questions on the final:

- Given a CFG G:
  - Remove left recursion
  - Left Factor
  - Find First/Follow sets for each non-terminal
  - Create a LL(1) Parse Table

FR-195: Questions on Final

You can expect to find the following types of questions on the final:

- Given a CFG G:
  - Create LR(0) states and transitions
  - Create LR(0) parse table
  - Find First/Follow sets for each non-terminal, create SLR(1) parse tabe

FR-196: Questions on Final

You can expect to find the following types of questions on the final:

- Given a **new** statement / expression type:
  - Create a **new** AST node (like Pascal For loops)
  - Given a piece of simpleJava code, give the AST that represents that piece of code

FR-197: Questions on Final

You can expect to find the following types of questions on the final:

- Given a piece of simpleJava code
  - Give the AAT that represents that piece of code
  - Tile the AAT, by drawing a circle around each tile. Each tile should be implemented by a single assembly language instruction, plus any extra instructions that are necessary for stack maintenance.
  - Show the generated code, noting which tile each line of assembly came from

FR-198: Questions on Final

You can expect to find the following types of questions on the final:

- Given a new assembly language instruction:
  - Create a tileset for that assembly language instruction, showing which code is associated with each tile
  - Give simpleJava code that uses one (or more) of these tiles
  - Show the assembly tree for the simpleJava code, circle tiles, and show final assembly language

FR-199: Questions on Final

You can expect to find the following types of questions on the final:

- Thinking / Synthesis question
  - Use information in slightly new ways
  - A little beyond “turn the crank”