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** $\Sigma$: Set of all possible symbols (characters) in the input file
  - Think of $\Sigma$ 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** $\epsilon$: 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 $L_2$, $L_1L_2 = \{xy|x \in L_1, y \in L_2\}$

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 $\Sigma$:

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

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\|$\epsilon$) | all strings over {a,b} that do not contain aa |

FR-10: **r.e. Shorthand**

| | | |
|---|---|---|
| [abcd] | = | (a\|b\|c\|d) |
| [b-g] | = | [befg] = (b\|e\|f\|g) |
| [b-gM-O] | = | [befgMNO] = (b\|e\|f\|g\|M\|N\|O) |
| M? | = | (M \| $\epsilon$) |
| M+ | = | MM* |
| . | = | Any character except newline |
| ”vc” | = | The string vc exactly |
| 4”.”5 | = | String 4.5 (not 4¡any¿5) |

FR-11: **r.e. Shorthand Examples**

| Regular Expression | Langauge |
|---:|:---|
| if | {if} |
| [a-z][0-9a-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 |

FR-12: **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> → <Terminals & Non-Terminals>
- Special Non-Terminal – Initial Symbol

FR-14: **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

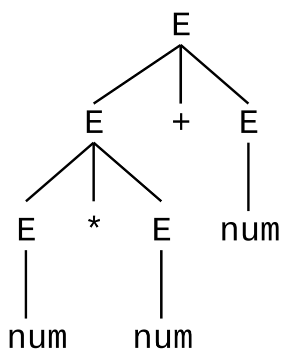
  Until all elements in the string are terminals

FR-15: **CFG Example**

$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow E * E$
$E \rightarrow E/E$
$E \rightarrow$ 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 $\epsilon$, if $S$ can produce $\epsilon$)

| | |
|---|---|
| $S' \rightarrow S$\$ | 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 $\epsilon$, if $S$ can produce $\epsilon$)

| | |
|---|---|
| $S' \rightarrow S$\$ | First($S'$) = {e, f, g, h} |
| $S \rightarrow AB$ | First($S$) = {e, f, g, h} |
| $S \rightarrow C$h | First($A$) = {e, $\epsilon$} |
| $A \rightarrow$ 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($\gamma$) 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.

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

FR-27: **Follow Sets**

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

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

FR-28: **Calculating Follow sets**

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

FR-29: **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 $S \to \gamma$ goes in row $S$, in all columns of First($\gamma$).
  - If First($\gamma$) contains $\epsilon$, then the rule $S \to \gamma$ 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: **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-32: **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-33: **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>)

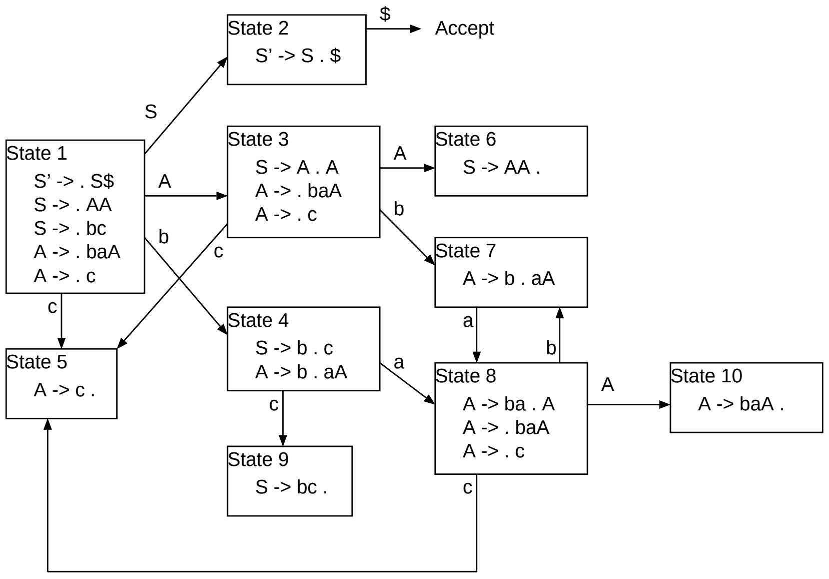
LR(0) parsers are the simplest of the LR parsers FR-34: **LR(0) Items**

- 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-35: **LR(0) States & Transitions**

(0) $S' \to S\$$
(1) $S \to AA$
(2) $S \to$ bc
(3) $S \to$ ba$A$
(4) $A \to$ c

FR-36: **LR(0) States & Transitions**



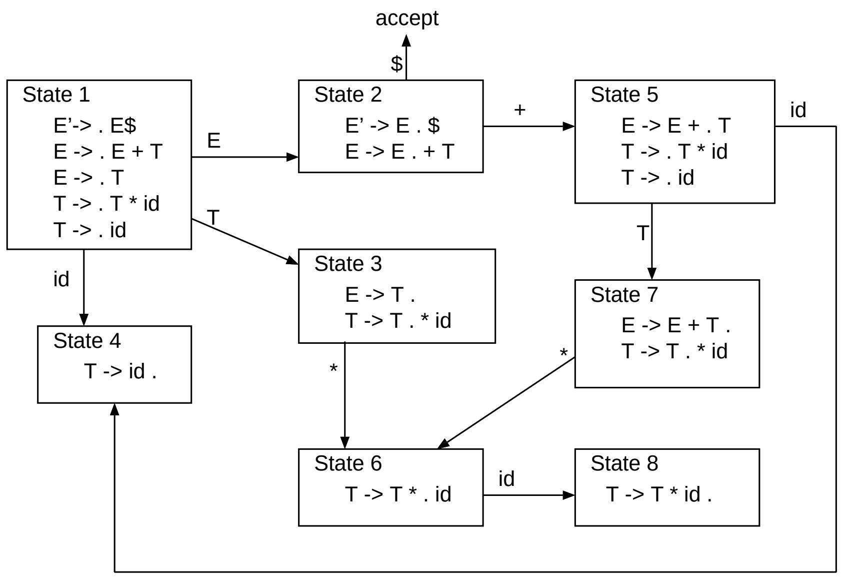
FR-37: **LR(0) Parse Table**

| | 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) | | |

FR-38: **Another LR(0) Example**

(0) $E' \to E\$$
(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T$ * id
(4) $T \to$ id

FR-39: **LR(0) States & Transitions**



FR-40: **LR(0) Parse Table**

| | id | + | * | $ | $E$ | $T$ |
|---|---|---|---|---|---|---|
| 1 | s4 | | | | g2 | g3 |
| 2 | | s5 | | accept | | |
| 3 | r(2) | r(2) | r(2),s6 | r(2) | | |
| 4 | r(4) | r(4) | r(4) | r(4) | | |
| 5 | s4 | | | | | g7 |
| 6 | s8 | | | | | |
| 7 | r(1) | r(1) | r(1),s6 | r(1) | | |
| 8 | r(3) | r(3) | r(3) | r(3) | | |

FR-41: **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*

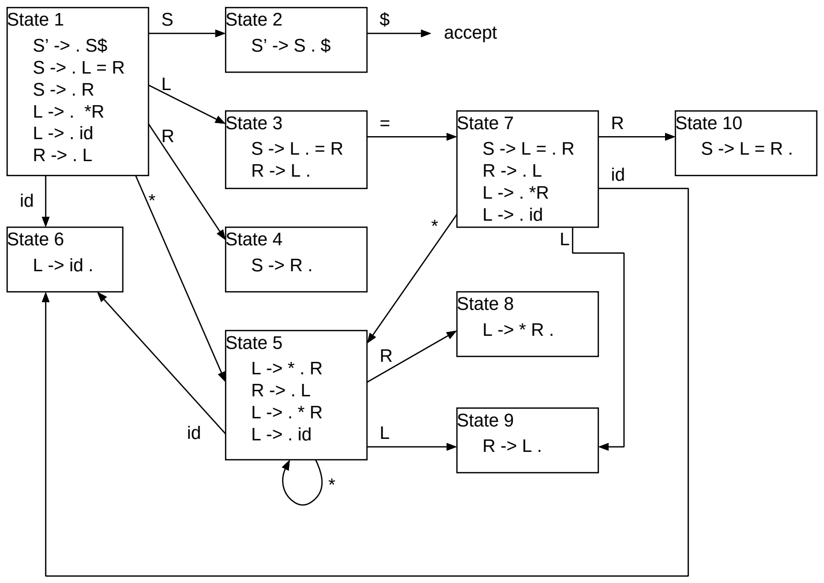
FR-42: **SLR(1) Parse Table**

| | id | + | * | $ | $E$ | $T$ |
|---|---|---|---|---|---|---|
| 1 | s4 | | | | g2 | g3 |
| 2 | | s5 | | accept | | |
| 3 | | r(2) | s6 | r(2) | | |
| 4 | | r(4) | r(4) | r(4) | | |
| 5 | s4 | | | | | g7 |
| 6 | s8 | | | | | |
| 7 | | r(1) | s6 | r(1) | | |
| 8 | | r(3) | r(3) | r(3) | | |

FR-43: **Yet Another Example**

(0) $S' \rightarrow S\$$
(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow {*}R$
(4) $L \rightarrow$ id
(5) $R \rightarrow L$

FR-44: **LR(0) States & Transitions**



FR-45: **SLR(1) Parse Table**

| | id | = | * | $ | $S$ | $L$ | $R$ |
|---|---|---|---|---|---|---|---|
| 1 | s6 | | s5 | | g2 | g3 | g4 |
| 2 | | | | accept | | | |
| 3 | | r(5),s7 | | r(5) | | | |
| 4 | | | | r(2) | | | |
| 5 | s6 | | s5 | | | g9 | g8 |
| 6 | | r(4) | | r(4) | | | |
| 7 | s6 | | s5 | | | g9 | g10 |
| 8 | | r(3) | | r(3) | | | |
| 9 | | r(5) | | r(5) | | | |
| 10 | | | | r(1) | | | |

FR-46: **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

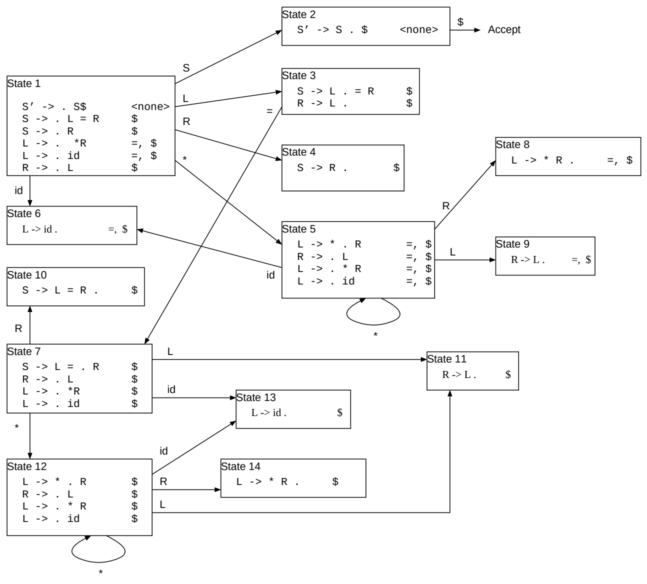
FR-47: **LR(1) Example**

(0) $S' \rightarrow X\$$
(1) $X \rightarrow CX$
(2) $X \rightarrow \epsilon$
(3) $C \rightarrow$ ab
(4) $C \rightarrow$ b

FR-48: **LR(1) Example**

(0) $S' \rightarrow S\$$
(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow *R$
(4) $L \rightarrow \text{id}$
(5) $R \rightarrow L$

FR-49: **LR(1) States and Transitions**



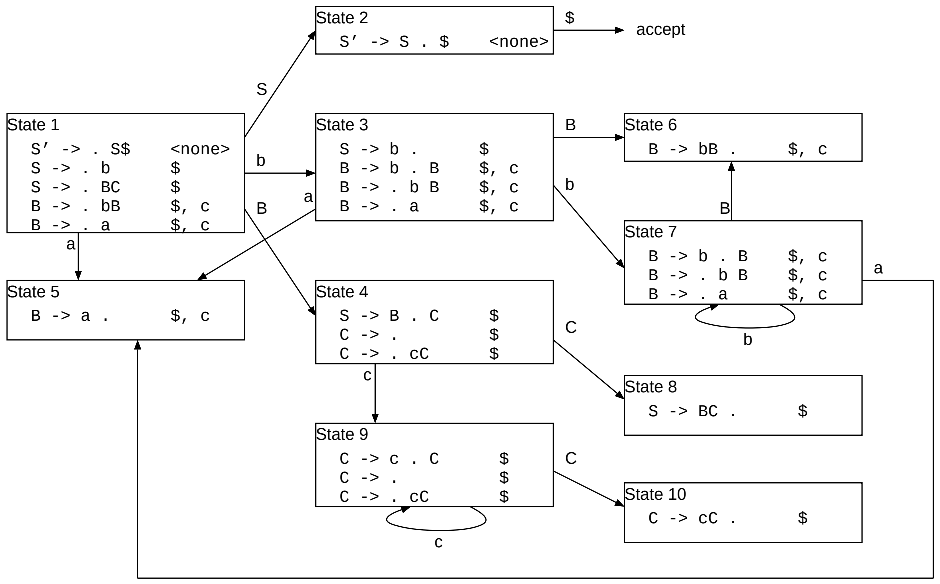
FR-50: **LR(1) Parse Table**

| | id | = | * | $ | $S$ | $L$ | $R$ |
|---|---|---|---|---|---|---|---|
| 1 | s6 | | s5 | | g2 | g3 | g4 |
| 2 | | | | accept | | | |
| 3 | | s7 | | r(5) | | | |
| 4 | | | | r(2) | | | |
| 5 | s6 | | s5 | | | g9 | g8 |
| 6 | | r(4) | | r(4) | | | |
| 7 | s13 | | s12 | | | g11 | g10 |
| 8 | | r(3) | | r(3) | | | |
| 9 | | r(5) | | r(5) | | | |
| 10 | | | | r(1) | | | |
| 11 | | | | r(5) | | | |
| 12 | s13 | | s12 | | | g11 | g14 |
| 13 | | | | r(4) | | | |
| 14 | | r(3) | | r(3) | | | |

FR-51: **More LR(1) Examples**

(0) $S' \rightarrow S\$$
(1) $S \rightarrow BC$
(2) $S \rightarrow \text{b}$
(3) $B \rightarrow \text{b}B$
(4) $B \rightarrow \text{a}$
(5) $C \rightarrow \epsilon$
(6) $C \rightarrow \text{c}C$

FR-52: **LR(1) States & Transitions**



FR-53: **LR(1) Parse Table**

| | a | b | c | $ | $S$ | $B$ | $C$ |
|---|---|---|---|---|---|---|---|
| 1 | s5 | s3 | | | g2 | g4 | |
| 2 | | | | accept | | | |
| 3 | s5 | s7 | | r(2) | | g6 | |
| 4 | | | s9 | r(5) | | | g8 |
| 5 | | | r(4) | r(4) | | | |
| 6 | | | r(3) | r(3) | | | |
| 7 | s5 | s7 | | | | g6 | |
| 8 | | | | r(1) | | | |
| 9 | | | s9 | | | | g10 |
| 10 | | | | r(6) | | | |

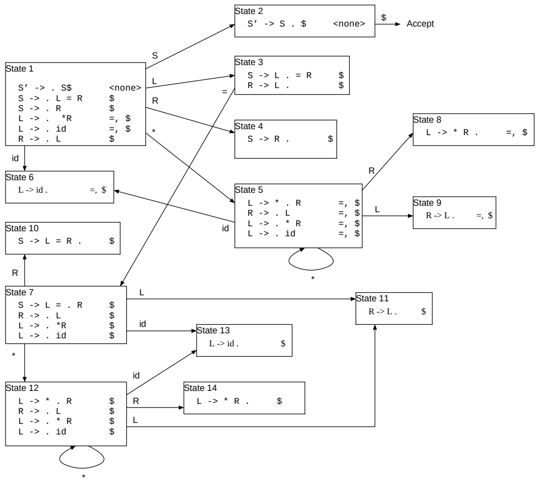
FR-54: **LALR Parsers**

- LR(1) Parsers are more powerful than LR(0) or SLR(1) parsers
- LR(1) Parsers can have many more states than LR(0) or SLR(1) parsers
  - My simpleJava implementation has 139 LR(0) states, and *thousands* of LR(1) states
- We'd like *nearly* the power of LR(1), with the memory requirements of LR(0)

FR-55: **LALR Parsers**

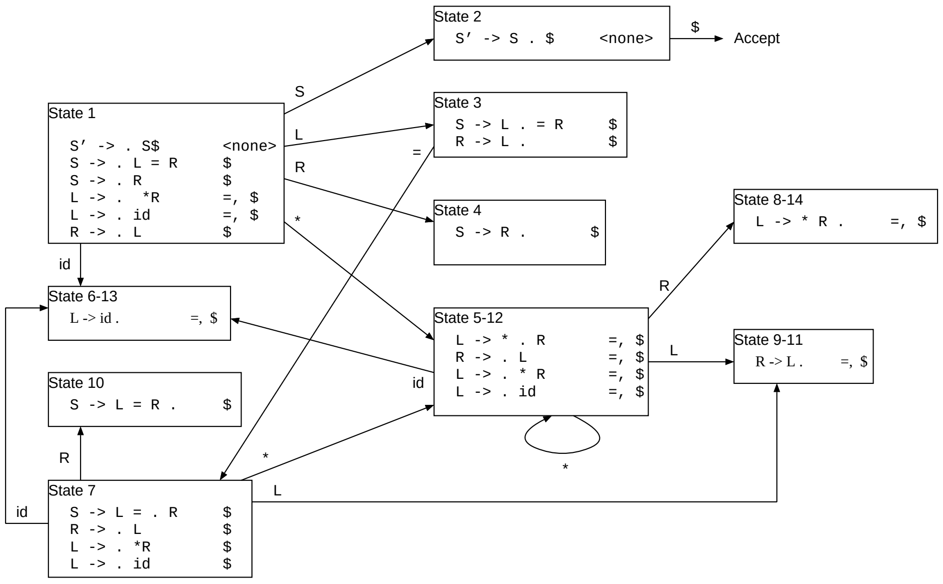
- LR(1) parsers can have large numbers of states
- Many of the states will be nearly the same – they will differ only in Lookahead
- IDEA – Combine states that differ only in lookahead values
  - Set lookahead of combined state to union of lookahead values from combining states

FR-56: **LALR Parser Example**



Can combine 5 & 12, 6 & 13, 8 & 14, 9 & 11

FR-57: **LALR Parser Example**



FR-58: **LALR Parser Example**

| | id | = | * | $ | $S$ | $L$ | $R$ |
|---|---|---|---|---|---|---|---|
| 1 | s6-13 | | s5-12 | | g2 | g3 | g4 |
| 2 | | | | accept | | | |
| 3 | | s7 | | r(5) | | | |
| 4 | | | | r(2) | | | |
| 5-12 | s6-13 | | s5-12 | | | g9-11 | g8-14 |
| 6-13 | | r(4) | | r(4) | | | |
| 7 | s6-13 | | s5-12 | | | g9-11 | g10 |
| 8-14 | | r(3) | | r(3) | | | |
| 9-11 | | r(5) | | r(5) | | | |
| 10 | | | | r(1) | | | |

FR-59: **More LALR Examples**

(0) $S' \rightarrow S\$$
(1) $S \rightarrow A$a
(2) $S \rightarrow$b$A$c
(3) $S \rightarrow B$c
(4) $S \rightarrow$b$B$a
(5) $A \rightarrow$d
(6) $B \rightarrow$d

FR-60: **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-61: **Parse Tree Example**

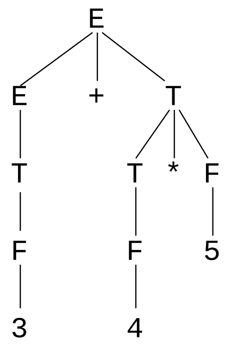
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow$ num

Parse tree for 3 + 4 * 5

FR-62: **Parse Tree Example**

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow$ num

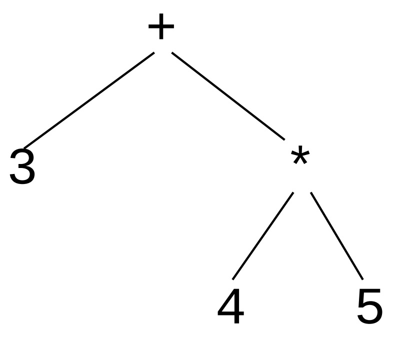
Parse tree for 3 + 4 * 5



FR-63: **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-64: **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)

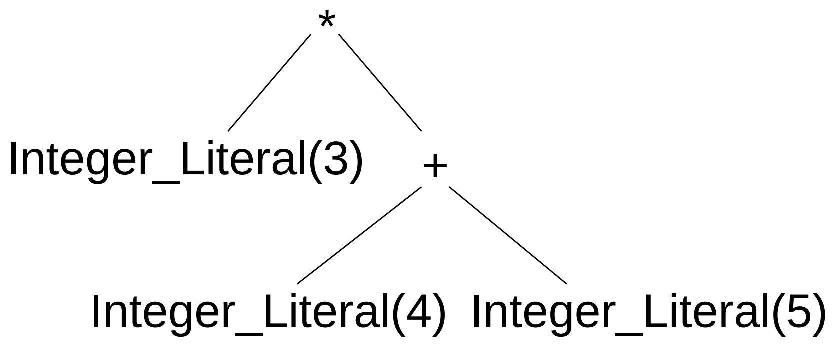
FR-65: **AST – Expressions**

- 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-66: **AST – Expressions**

3 * (4 + 5)



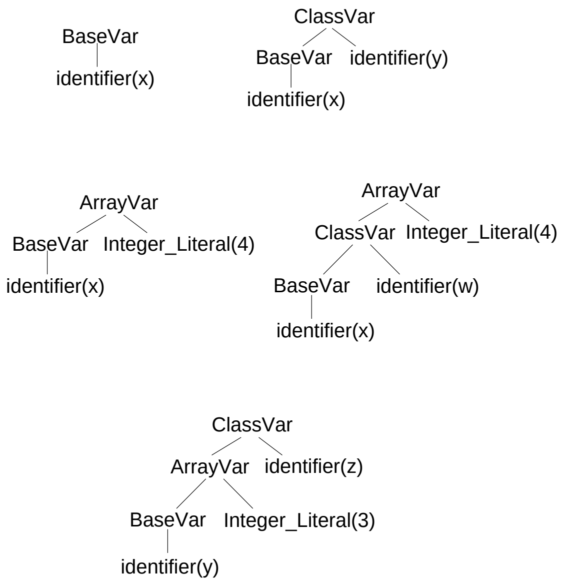
FR-67: **AST – Variables**

- 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-68: **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-69: **AST – Variables**



FR-70: **AST – Instance Variables**

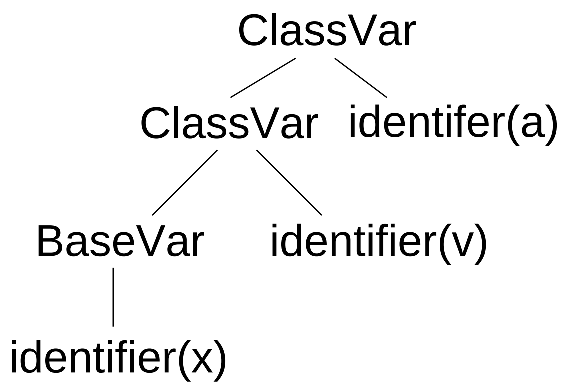
```
class simpleClass {
   int a;
   int b;
}
class complexClass {
   int u;
   simpleClass v;
}
void main() {
   complexClass x;
   x = new complexClass();
   x.v = new simpleClass();

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

FR-71: **AST – Instance Variables**

x.v.a

FR-72: **AST – Instance Variables**

x.v.a

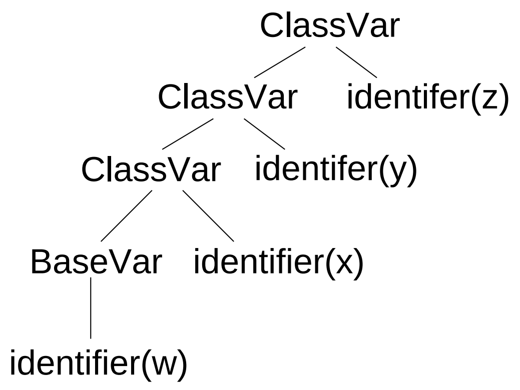


FR-73: **AST – Instance Variables**

w.x.y.z

FR-74: **AST – Instance Variables**

w.x.y.z

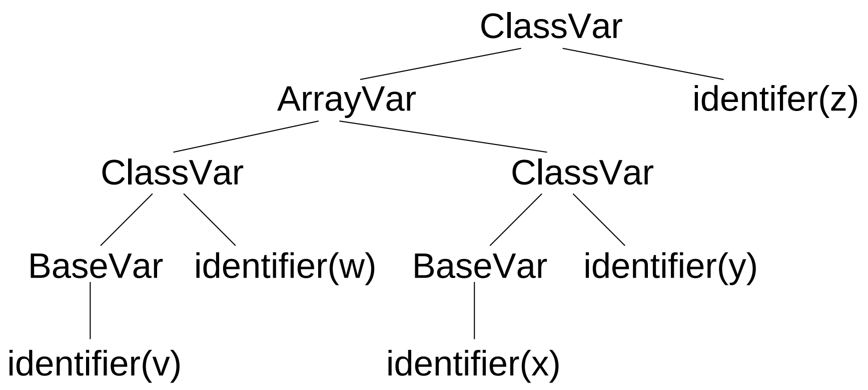


FR-75: **AST – Instance Variables**

v.w[x.y].z

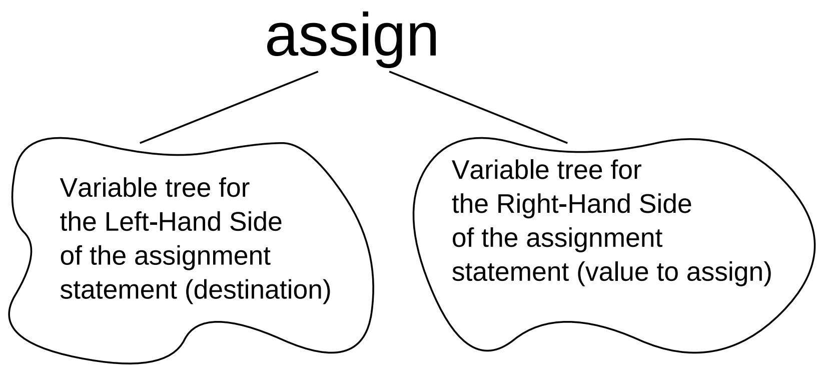
FR-76: **AST – Instance Variables**

v.w[x.y].z



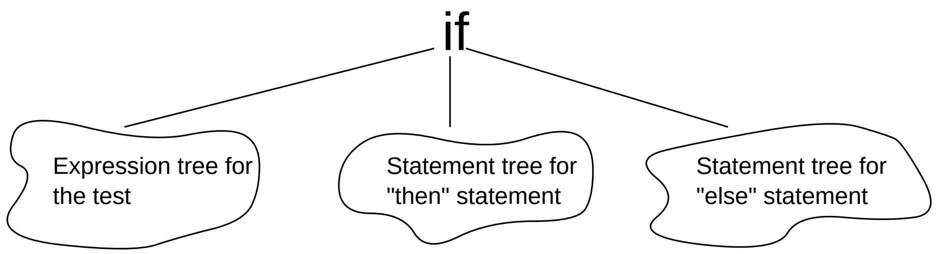
FR-77: **AST – Statements**

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


assign
Variable tree for the Left-Hand Side of the assignment statement (destination)
Variable tree for the Right-Hand Side of the assignment statement (value to assign)

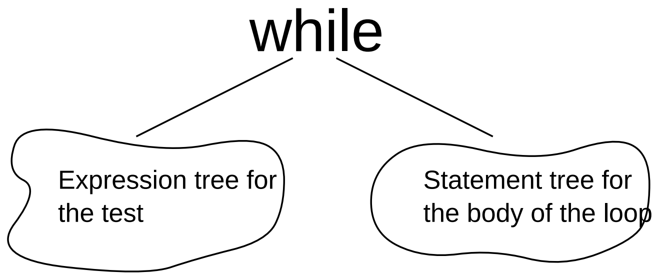
FR-78: **AST – Statements**

- **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.


if
Expression tree for the test
Statement tree for "then" statement
Statement tree for "else" statement

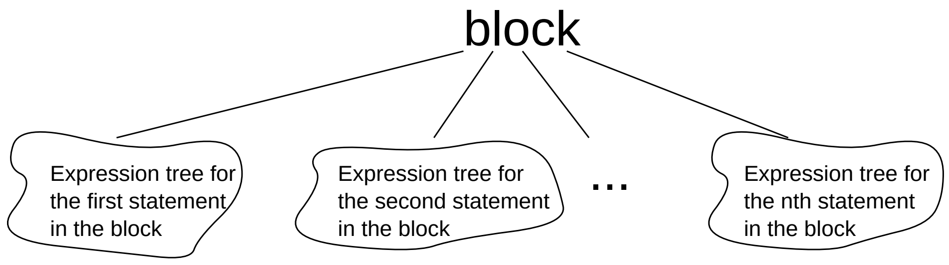
FR-79: **AST – Statements**

- **While Statement** Root is "While", children for test and body of the while loop.


while
Expression tree for the test
Statement tree for the body of the loop

FR-80: **AST – Statements**

- **Block Statement** That is, {<statement1>; <statement2>; ... <statementn> }. Block statements have a variable number of children, represented as a linked list of children.


block
Expression tree for the first statement in the block
Expression tree for the second statement in the block
...
Expression tree for the nth statement in the block

FR-81: **Syntax Errors/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-82: **Syntax Errors/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-83: **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-84: **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-85: **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-86: **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-87: **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-88: **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-89: **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-90: **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

FR-91: **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-92: **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-93: **Environments & Name Spaces**

- Types and variables have different name spaces in simpleJava, C, and standard Java:

```
simpleJava:

class foo {
   int foo;
}

void main() {
   foo foo;
   foo = new foo();
   foo.foo = 4;
   print(foo.foo);
}
```

FR-94: **Environments & Name Spaces**

- Types and variables have different name spaces in simpleJava, C, and standard Java:

```
C:
#include <stdio.h>

typedef int foo;
  int main() {
  foo foo;
  foo = 4;
  printf("%d", foo);
  return 0;
}
```

FR-95: **Environments & Name Spaces**

- Types and variables have different name spaces in simpleJava, C, and standard Java:

```
Java:

class EnviornTest {

   static void main(String args[]) {

      Integer Integer = new Integer(4);
      System.out.print(Integer);
    }
}
```

FR-96: **Environments & Name Spaces**

- Variables and functions in C share the same name space, so the following C code is **not** legal:

```
int foo(int x) {
  return 2 * x;
}

int main() {
  int foo;
  printf("%d\n",foo(3));
  return 0;
}
```

- The variable definition `int foo;` masks the function definition for `foo`

FR-97: **Environments & Name Spaces**

- Both standard Java and simpleJava use different name spaces for functions and variables
- Defining a function and variable with the same name will not confuse Java or simpleJava in the same way it will confuse C
    - *Programmer* might still get confused ...

FR-98: **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-99: **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-100: **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-101: **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-102: **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-103: **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-104: **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-105: **Stack Implementation**

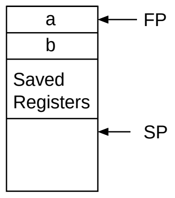
- Stack is implemented with two registers:
  - Frame Pointer (FP) points to the beginning of the current stack. frame
  - Stack Pointer (SP) points to the next free address on the stack.
- Stacks grow from large addresses to small addresses.

FR-106: **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-107: **Stack Frames**

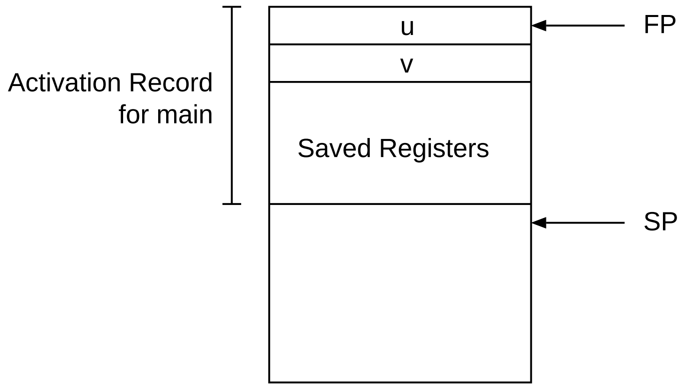
```
int foo() {
   int a;
   int b;

   /* body of foo */
}
```



FR-108: **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);
}
```

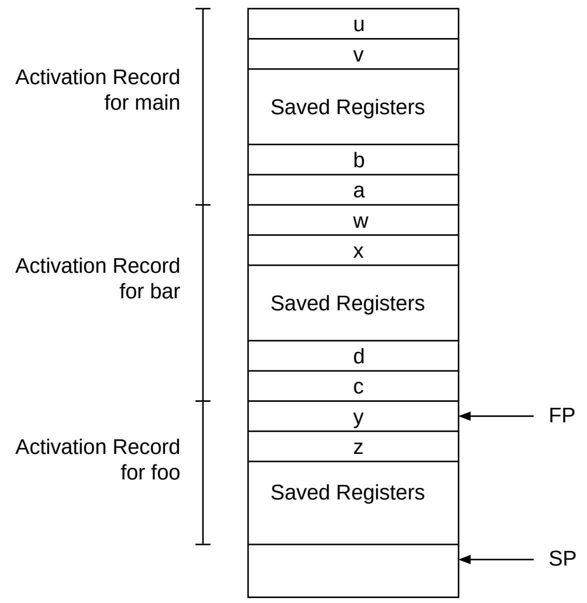
FR-109: **Stack Frames**



FR-110: **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);
}
```

FR-111: **Stack Frames**


Activation Record for main
Activation Record for bar
Activation Record for foo
u
v
Saved Registers
b
a
w
x
Saved Registers
d
c
y
z
Saved Registers
FP
SP

FR-112: **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-113: **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-114: **Abstract Assembly**

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

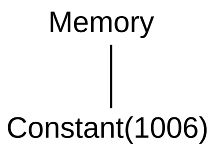
FR-115: **Expression Trees**

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

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

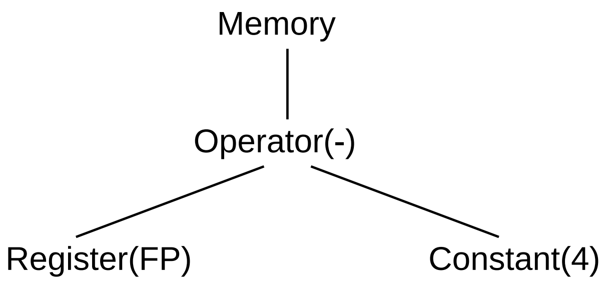
FR-116: **Expression Trees**

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



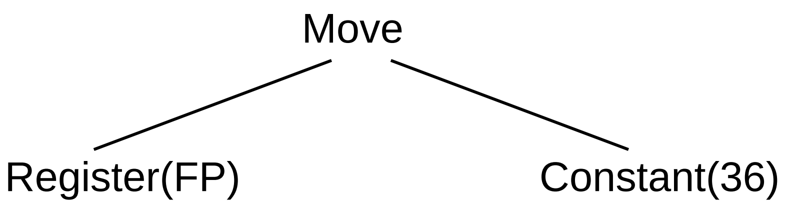
FR-117: **Expression Trees**

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



FR-118: **Statement Trees**

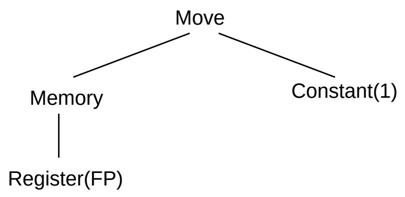
- Move Statements
  - Move statements are used to move data into either a memory location or a register
  - Left subtree of a move statement must be a register or memory expression
  - Right subtree of a move statement is any expression
  - To store the value 36 in the Frame Pointer:



FR-119: **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:

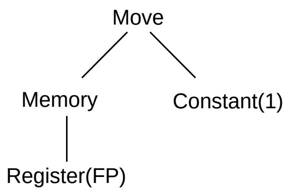


FR-120: **Abstract Assembly Examples**

```
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-121: **Abstract Assembly Examples**

```
x = 1;
```

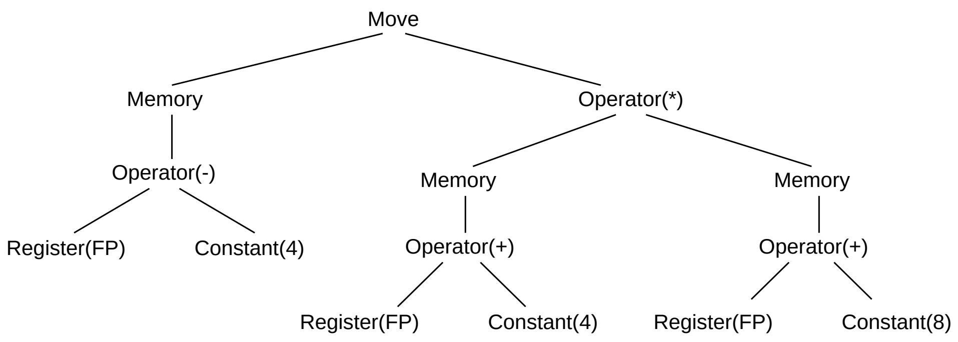


FR-122: **Abstract Assembly Examples**

```
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-123: **Abstract Assembly Examples**

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

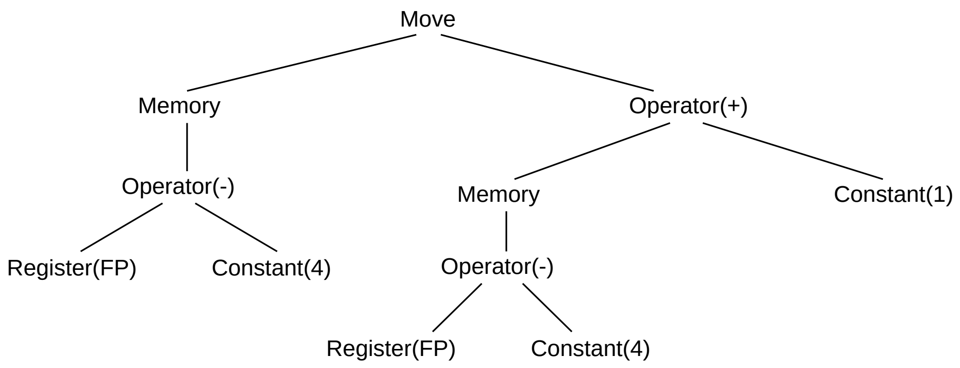


FR-124: **Abstract Assembly Examples**

```
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;
}
```

FR-125: **Abstract Assembly Examples**

```
y++;
```

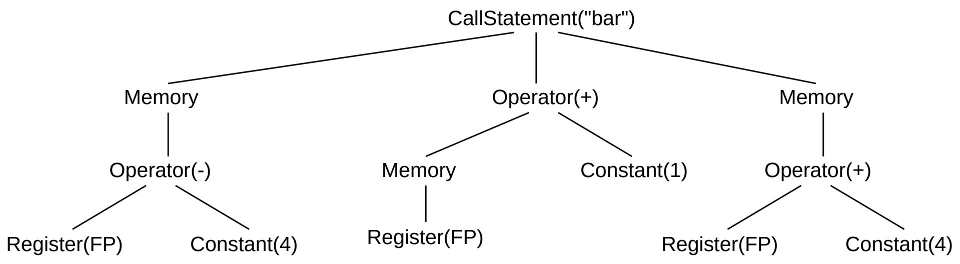


FR-126: **Abstract Assembly Examples**

```
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;
}
```

FR-127: **Abstract Assembly Examples**

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

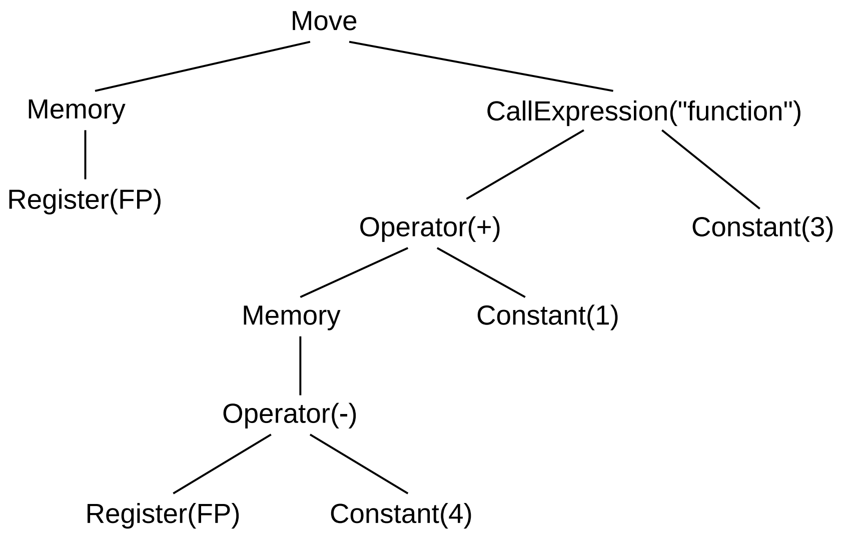


FR-128: **Abstract Assembly Examples**

```
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-129: **Abstract Assembly Examples**

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

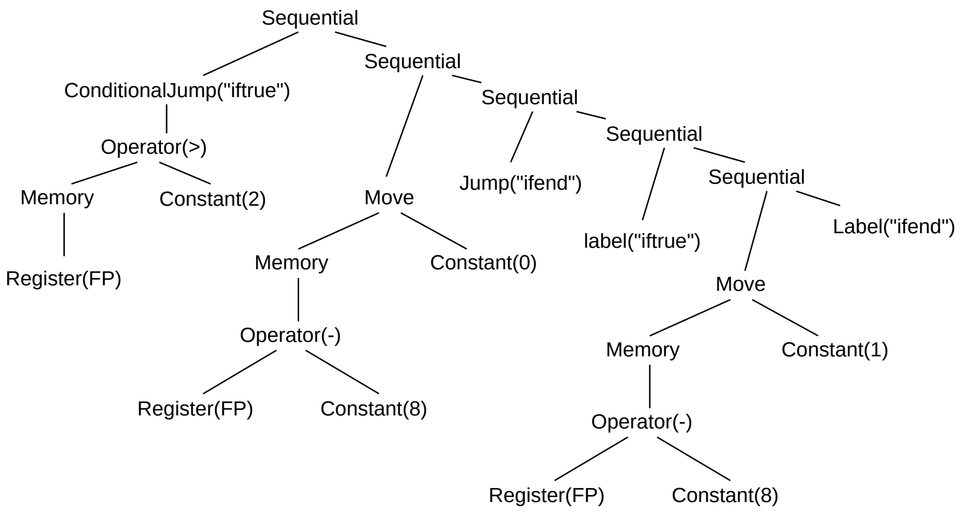


FR-130: **Abstract Assembly Examples**

```
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;        -|
}
```

FR-131: **Abstract Assembly Examples**

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

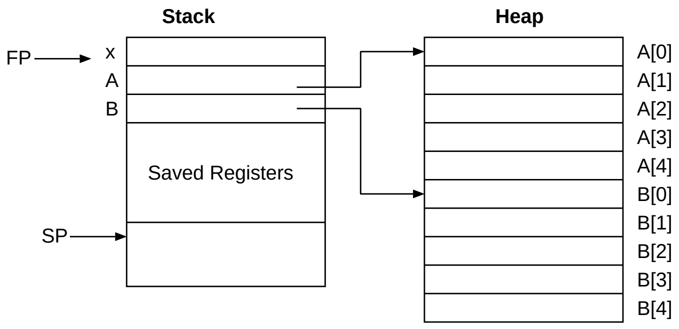


FR-132: **Creating Abstract Assembly**

- 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-133: **Array Variables**

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



FR-134: **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

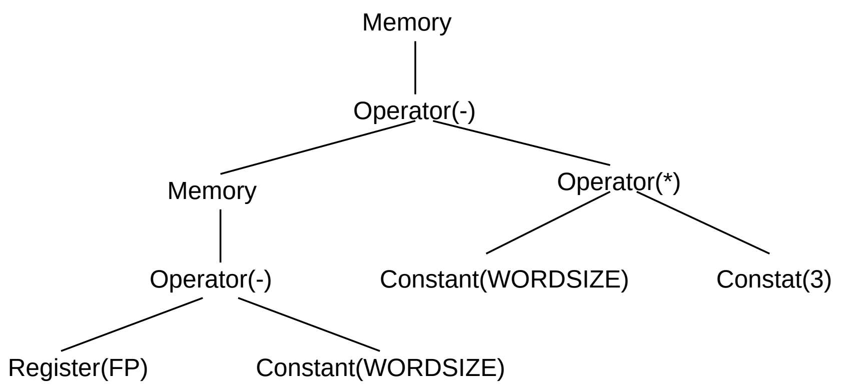
  - Dereference this pointer, to get the correct memory location

FR-135: **Array Variables**

- `A[3]`

FR-136: **Array Variables**

- `A[3]`

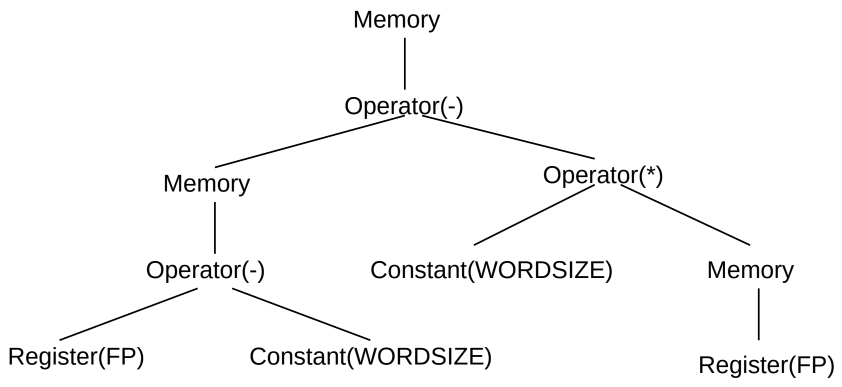


FR-137: **Array Variables**

- `A[x]`

FR-138: **Array Variables**

- `A[x]`

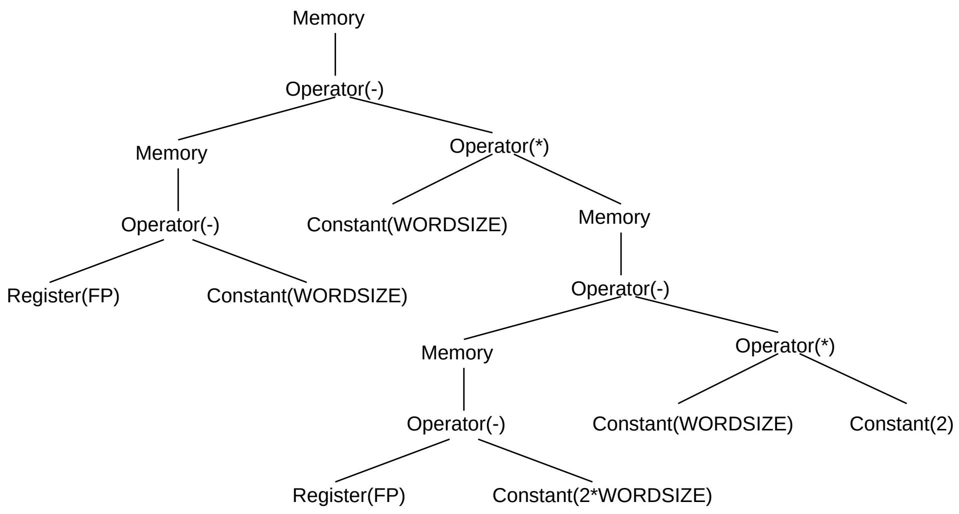


FR-139: **Array Variables**

- `A[B[2]]`

FR-140: **Array Variables**

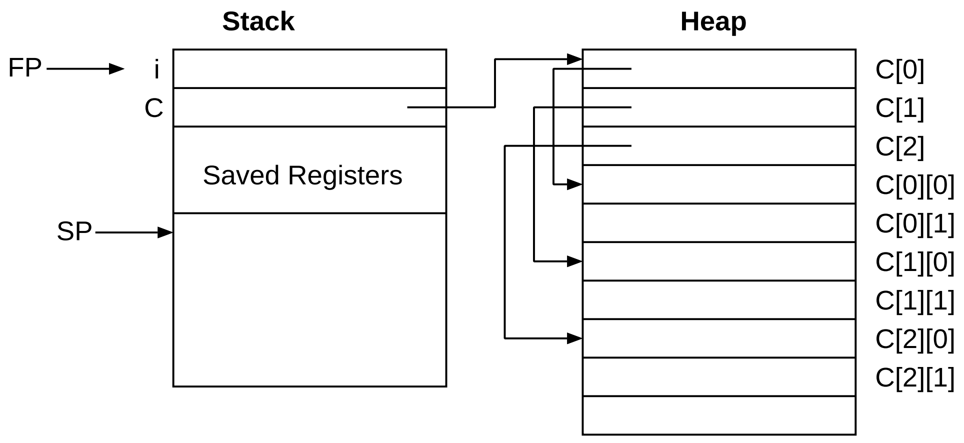
- `A[B[2]]`



FR-141: **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-142: **2D Arrays**

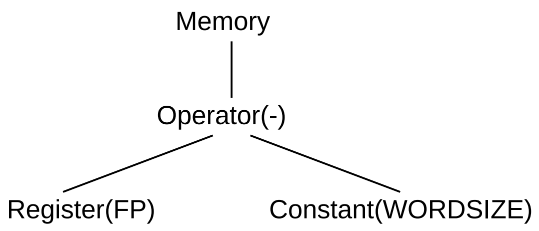


FR-143: **2D Arrays**

- C

FR-144: **2D Arrays**

- C

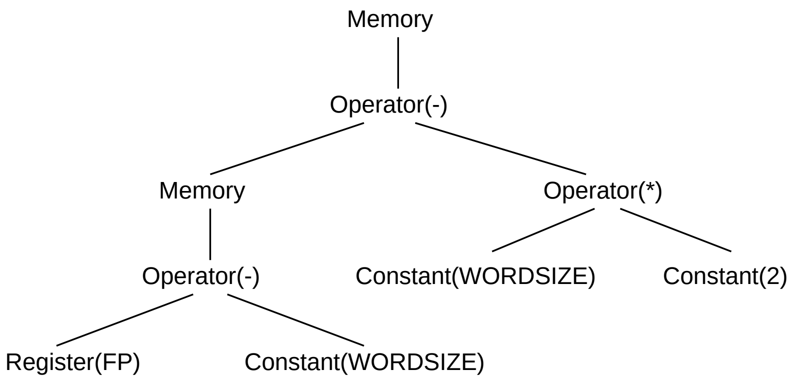


FR-145: **2D Arrays**

- C[2]

FR-146: **2D Arrays**

- C[2]

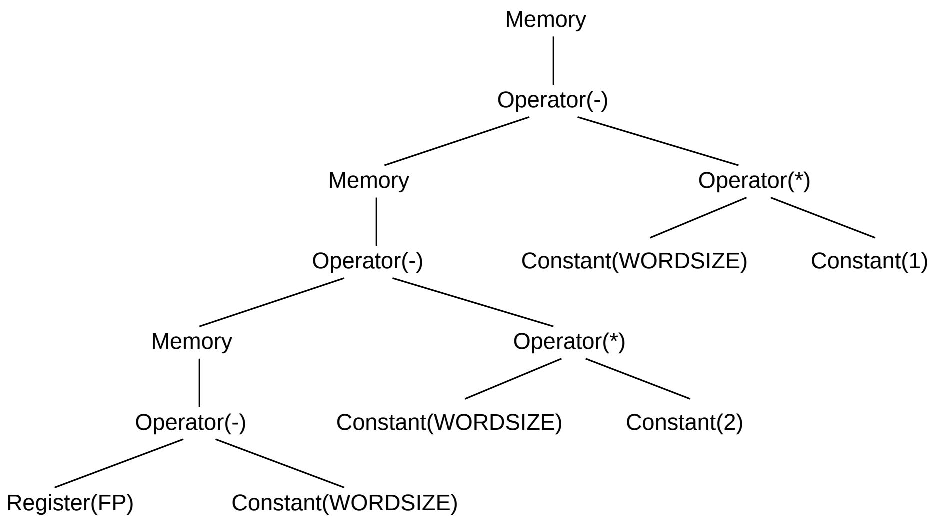


FR-147: **2D Arrays**

- C[2][1]

FR-148: **2D Arrays**

- C[2][1]



FR-149: **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-150: **Instance Variables**

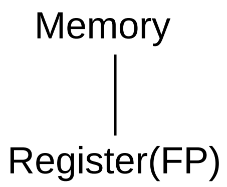
```
class simpleClass {
   int x;
   int y;
   int A[];
}

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

   /* Body of main */
}
```

FR-151: **Instance Variables**

- Variable `s`

FR-152: **Instance Variables**

- Variable `s`

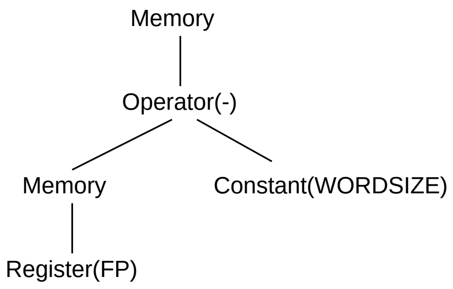


FR-153: **Instance Variables**

- Variable `s.y`

FR-154: **Instance Variables**

- Variable `s.y`

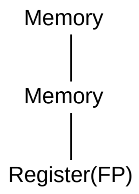


FR-155: **Instance Variables**

- Variable `s.x`

FR-156: **Instance Variables**

- Variable `s.x`

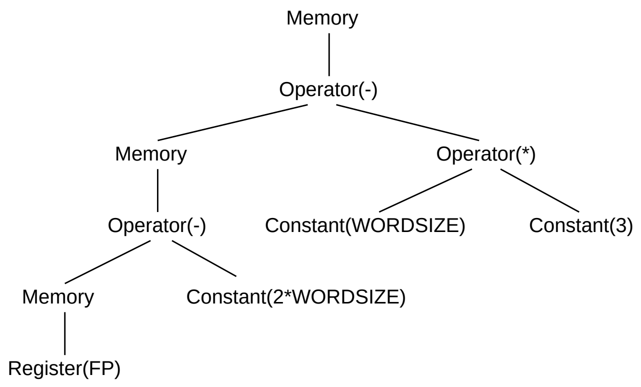


FR-157: **Instance Variables**

- Variable `s.A[3]`

FR-158: **Instance Variables**

- Variable `s.A[3]`

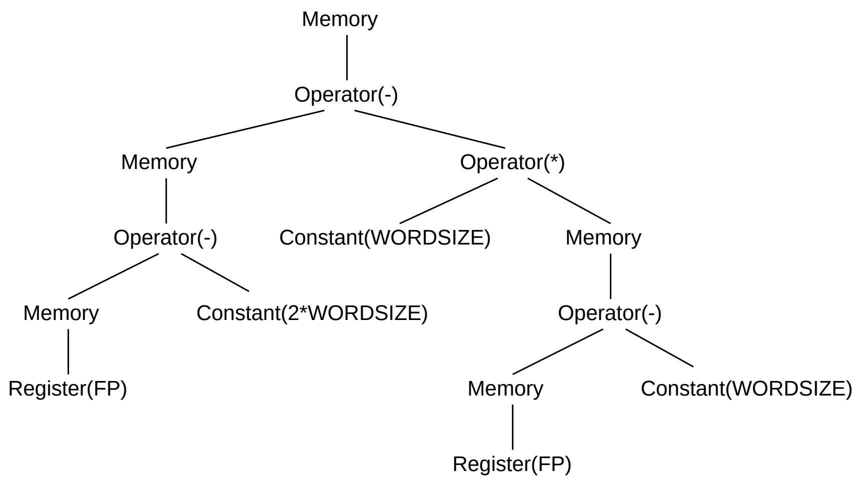


FR-159: **Instance Variables**

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

FR-160: **Instance Variables**

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



FR-161: **While Statements**

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

FR-162: **While Statements**

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

- Straightforward version:

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

FR-163: **While Statements**

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

- More Efficient:

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

FR-164: **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-165: **MIPS**

| Instruction | Description |
|---|---|
| lw rt, <offset> (base) | Add the constant value *<offset>* to the register *base* to get an address. Load the contents of this address into the register *rt*. rt = M[base + <offset>] |
| sw rt, <offset> (base) | Add the constant value <offset> to the register base to get an address. Store the contents of rt into this address. M[base + <offset>] = rt |
| add rd, rs, rt | Add contents of registers rs and rt, put result in register rd |

FR-166: **MIPS**

| Instruction | Description |
|---|---|
| sub rd, rs, rt | Subtract contents of register rt from rs, put result in register rd |
| addi rt, rs, <val> | Add the constant value <val> to register rs put result in register rt |
| mult rs, rt | Multiply contents of register rs by register rt, put the low order bits in register LO, and the high bits in register HI |
| div rs, rt | Divide contents of register rs by register rt, put the quotient in register LO, and the remainder in register HI |

FR-167: **MIPS**

| Instruction | Description |
|---|---|
| mflo rd | Move contents of the special register LOW into the register rd |
| j <target> | Jump to the assembly label <*target*> |
| jal <target> | Jump and link. Put the address of the next instruction in the *Return* register, and then jump to the address <target>. Used for function and procedure calls |
| jr rs | Jump to the address stored in register rs. Used in conjunction with jal to return from function and procedure calls |

FR-168: **MIPS**

| Instruction | Description |
|---|---|
| beq rs, rt, <target> | if rs = rt, jump to the label <target> |
| bne rs, rt, <target> | if rs $\neq$ rt, jump to the label <target> |
| blez rs, <target> | if rs $\leq$ 0, jump to label <target> |
| bgtz rs, <target> | if rs > 0, jump to label <target> |
| bltz rs, <target> | if rs < 0, jump to the label <target> |
| bgez rs, <target> | if rs $\geq$ 0, jump to the label <target> |

FR-169: **Registers**

- MIPS processors use 32 different registers
- We will only use a subset for this project
  - (Though you can increase the number of registers used for temporary values fairly easily)

FR-170: **Registers**

| Mnemonic Name | SPIM Name | Description |
|---|---|---|
| $FP | $fp | Frame Pointer – Points to the top of the current activation record |
| $SP | $sp | Stack Pointer – Used for the activation record (stack frame) stack |
| $ESP | | Expression Stack Pointer – The expression stack holds temporary values for expression evaluations |
| $result | $v0 | Result Register – Holds the return value for functions |

FR-171: **Registers**

| Mnemonic Name | SPIM Name | Description |
|---|---|---|
| $return | $ra | Return Register – Holds the return address for the current function |
| $zero | $zero | Zero Register – This register always has the value 0 |
| $ACC | $t0 $t4 | Accumulator Register – Used for calculating the value of expressions |
| $t1 | $t1 | General Purpose Register |
| $t2 | $t2 | General Purpose Register |
| $t3 | $t3 | General Purpose Register |

FR-172: **Expression Stack**

- Long expressions – a * b + c * d * foo(x) – will require us to store several temporary values

- Since expressions can be arbitrarily long, we will need to store an unlimited number of partial solutions
    - Can't always use registers – not enough of them
    - Use a stack instead

FR-173: **Expression Stack**

- For now, we will use an entirely different stack than the one for activation records
    - Make debugging easier
    - Later on, we can combine the stacks

FR-174: **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-175: **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-176: **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-177: **Expression Trees**

- Constant Expressions
    - Constant(5)
        - Push the constant 5 on the top of the expression stack

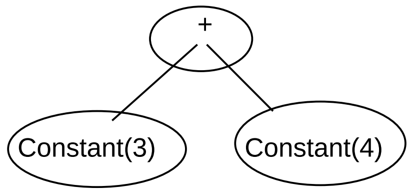
FR-178: **Expression Trees**

- Constant Expressions
    - How can we push the constant $x$ on top of the expression stack, using MIPS assembly?

```
addi $t1, $zero, x
sw   $t1, 0($ESP)
addi $ESP, $ESP, -4
```

FR-179: **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-180: **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-181: **Expression Trees**

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

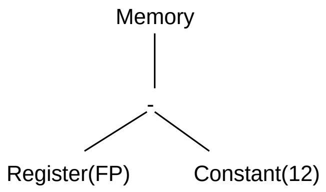
FR-182: **Expression Trees**

- Memory Accesses

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

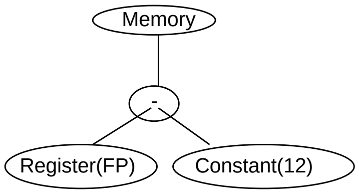
FR-183: **Expression Trees**

- Memory Example

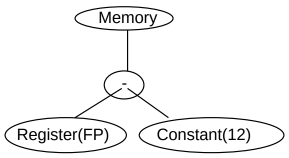


FR-184: **Expression Trees**

- Memory Example



FR-185: **Expression Trees**



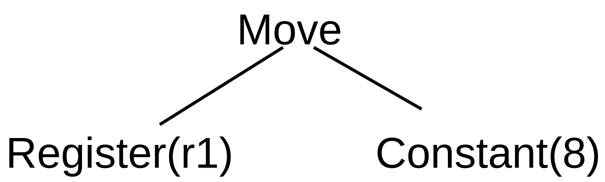
```
sw   $FP,  0($ESP)     % Store frame pointer on the top of the expressin stack
addi $ESP, $ESP,  -4   % Update the expression stack pointer
addi $t1,  $zero, 12   % Load the constant value 12 into the register $t1
sw   $t1,  0($ESP)     % Store $t1 on the top of the expression stack
addi $ESP, $ESP,  -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-186: **Simple Tiling**

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

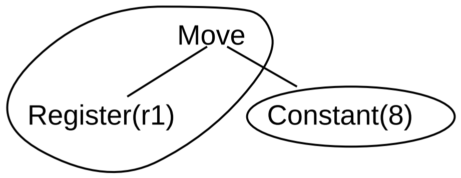
FR-187: **Statement Trees**

- Move Trees (Moving into Registers)



FR-188: **Statement Trees**

- Move Trees (Moving into Registers)

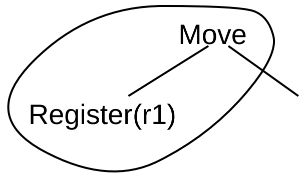


- The code for the MOVE tile needs to:
  - Pop the value to move off the stack

- Store the value in the appropriate register

FR-189: **Statement Trees**

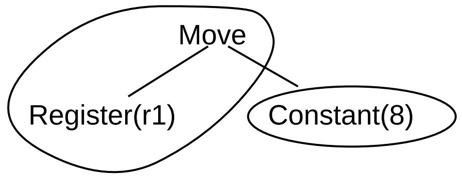
- Move Trees (Moving into Registers)



```
lw   $r1,  4($ESP)
addi $ESP, $ESP, 4
```

FR-190: **Statement Trees**

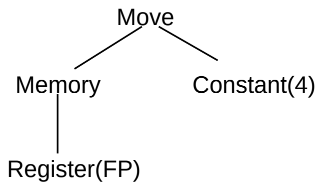
- Move Trees (Moving into Registers)



```
addi $t1,  $zero, 8
sw   $t1,  0($ESP)
addi $ESP, $ESP,  -4
lw   $r1,  4($ESP)
addi $ESP, $ESP, 4
```

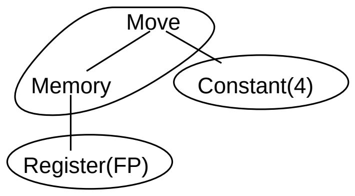
FR-191: **Statement Trees**

- Move Trees (Moving into MEMORY locations)



FR-192: **Statement Trees**

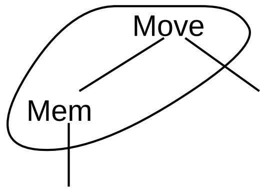
- Move Trees (Moving into MEMORY locations)



- 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

FR-193: **Statement Trees**

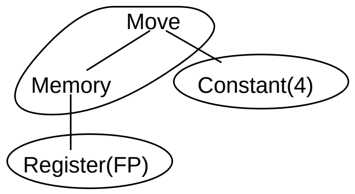
- Move Trees (Moving into MEMORY locations)



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

FR-194: **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-195: **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-196: **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-197: **Improved Tiling**

- Decrease reliance on the expression stack
  - Every expression is stored on the stack – even when we do not need to store partial results
  - Instead, we will only use the stack when we need to store partial results – and use registers otherwise

FR-198: **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

FR-199: **Accumulator Register**

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

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

FR-200: **Accumulator Register**

- Register trees
  - Tree: Register(r1)
  - Code:

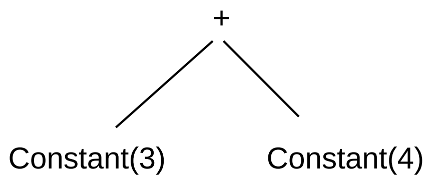
```
addi $ACC, $r1, 0
```

FR-201: **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

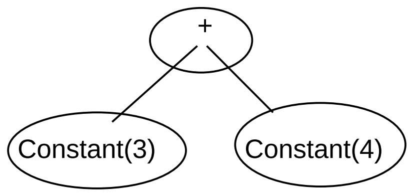
FR-202: **Accumulator Register**

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



FR-203: **Accumulator Register**

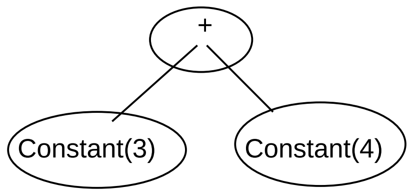
- Binary Operators (+, -, *, etc)



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

FR-204: **Accumulator Register**

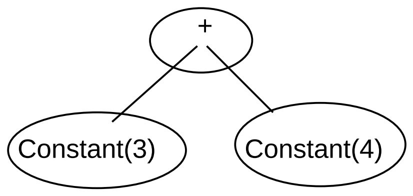
- Binary Operators (+, -, *, etc)



```
<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-205: **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
```

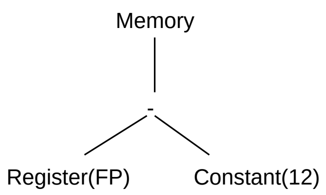
FR-206: **Accumulator Register**

- Memory Expression Trees
    - Code for a Memory tile:

```
lw $ACC, 0($ACC)
```

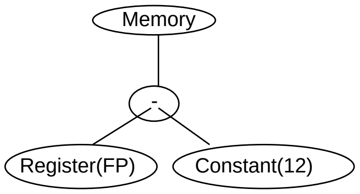
FR-207: **Accumulator Register**

- Memory Example



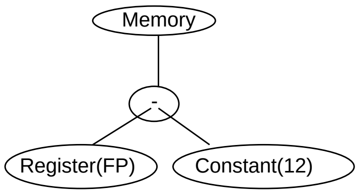
FR-208: **Accumulator Register**

- Memory Example



FR-209: **Accumulator Register**

- Memory Example



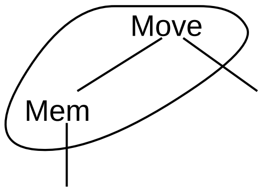
```
addi $ACC, $FP, 0         % Store the FP in the accumulator
sw   $ACC, 0($ESP)        % Store left operand on expression stack
addi $ESP, $ESP, -4       % Update expression stack pointer
addi $ACC, $zero, 12      % Store constant 4 in the accumulator
lw   $t1,  4($ESP)        % Pop left operand off expression stack
addi $ESP, $ESP, 4        % Update expression stack pointer
sub  $ACC, $t1, $ACC      % Do the addition
lw $ACC, 0($ACC)          % Dereference the ACC
```

FR-210: **Accumulator Register**

- Memory Move Statement Trees
  - Calculate the source & destination of the move
  - Like operator expressions, will need to store values on stack
  - Once source & destination are stored in registers, can use a `sw` statement

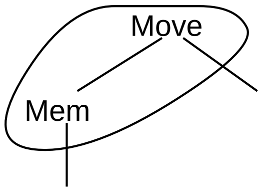
FR-211: **Accumulator Register**

- Memory Move Statement Trees



FR-212: **Accumulator Register**

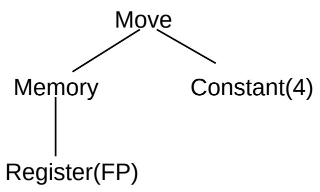
- Memory Move Statement Trees



```
<code for left subtree (destination)>
sw   $ACC,  0($ESP)
addi $ESP, $ESP,   -4
<code for right subtree (value to move)>
lw   $t1,  4($ESP)
addi $ESP, $ESP,   4
sw   $ACC, 0($t1)
```

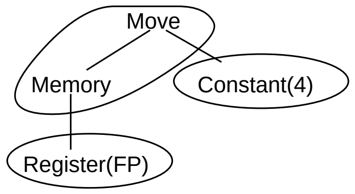
FR-213: **Accumulator Register**

- Memory Move Example



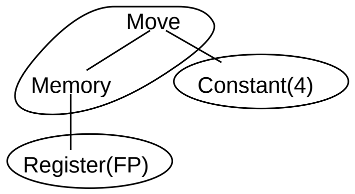
FR-214: **Accumulator Register**

- Memory Move Example



FR-215: **Accumulator Register**

- Memory Move Example



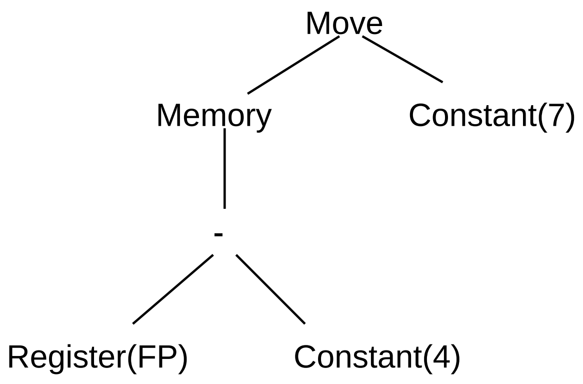
```
addi $ACC, $FP, 0        %  Code for Register(FP) tile
sw   $ACC,  0($ESP)      %  Store destination on expression stack
addi $ESP, $ESP,   -4    %  Update expression stack pointer
addi $ACC, $zero, 4      %  Code for Constant(4) tile
lw   $t1,  4($ESP)       %  Load destination into a register
addi $ESP, $ESP,   4     %  Update expression stack pointer
sw   $ACC, 0($t1)        %  Implement the move
```

FR-216: **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

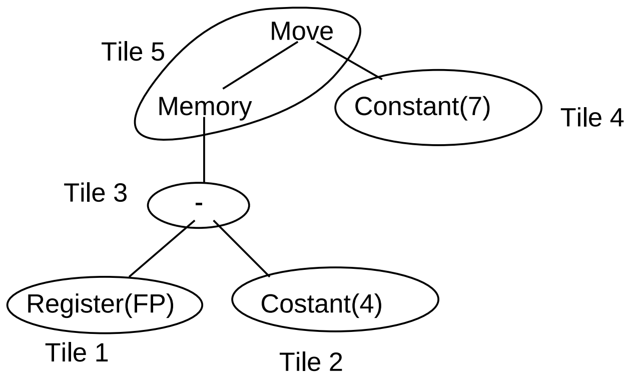
FR-217: **Larger Tiles Example**

- Memory Move Expression



FR-218: **Larger Tiles Example**

- Standard Tiling



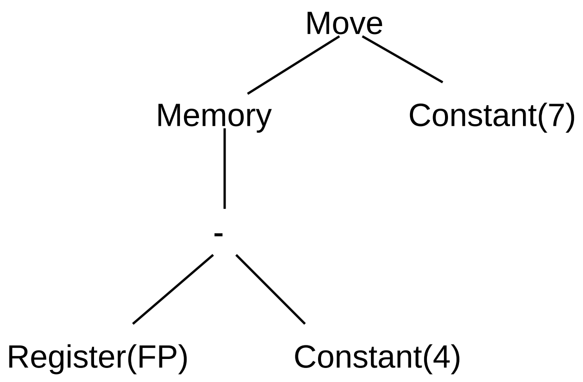
FR-219: **Larger Tiles Example**

- 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
```

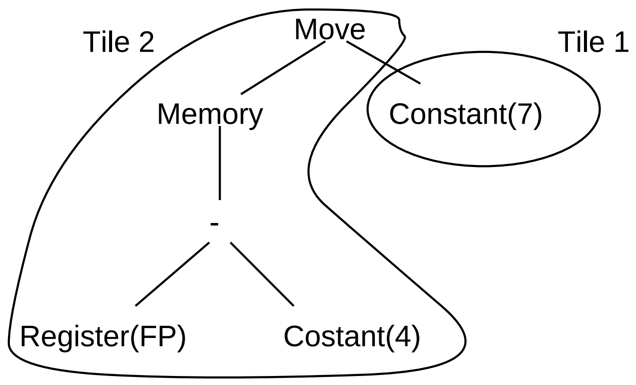
FR-220: **Larger Tiles Example**

- Memory Move Expression

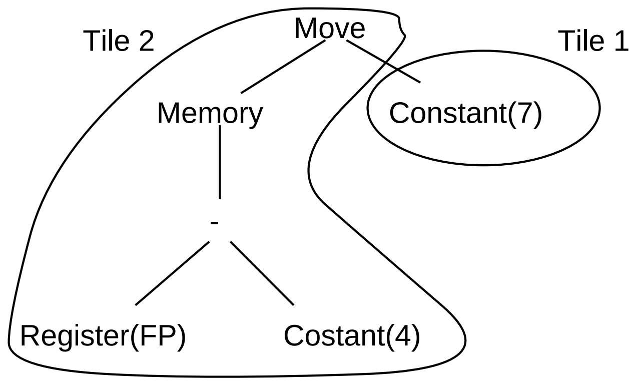


FR-221: **Larger Tiles Example**

- Using Larger Tiles

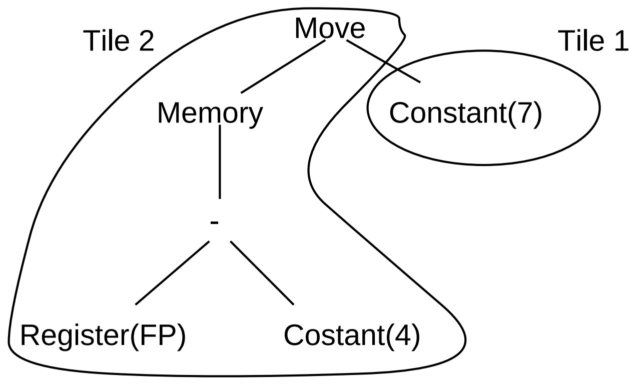


FR-222: **Larger Tiles Example**



- Code for Tile 2?

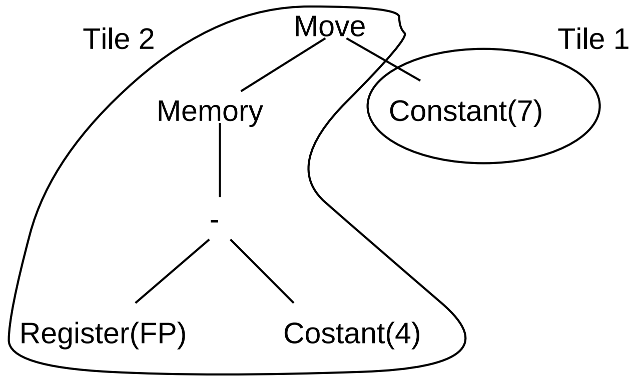
FR-223: **Larger Tiles Example**



- Code for Tile 2?

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

FR-224: **Larger Tiles Example**



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

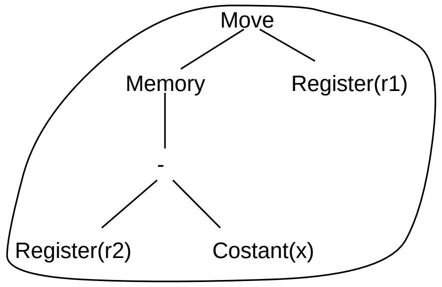
FR-225: **Larger Tiles**

- Can get a huge saving using larger tiles
- Especially if tile size is geared to functionality of the actual assembly
    - `sw` Stores the value in a register in a memory location pointed to by an offset off a different register
    - Tile that takes full advantage of this functionality will lead to efficient assembly

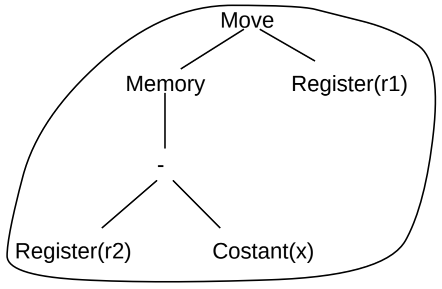
FR-226: **Larger Tiles**

- Design tiles based on the actual assembly language
- Take advantage of as many feature of the language as possible
- Create tiles that are as large as possible, that can be implemented with a single assembly language instruction
    - Plus some extra instructions to do stack maintenance

FR-227: **Larger Tiles – Examples**

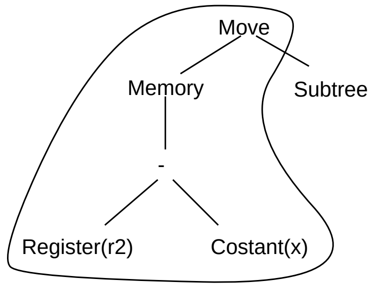


FR-228: **Larger Tiles – Examples**

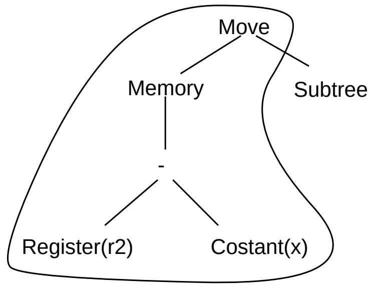


```
sw r1 -x(r2)
```

FR-229: **Larger Tiles – Examples**

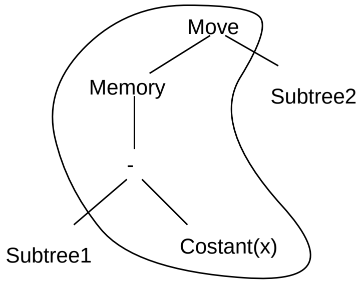


FR-230: **Larger Tiles – Examples**

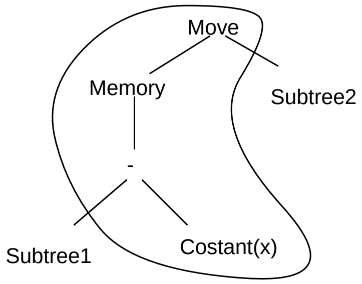


```
<code for Subtree>
sw $ACC, -x(r2)
```

FR-231: **Larger Tiles – Examples**



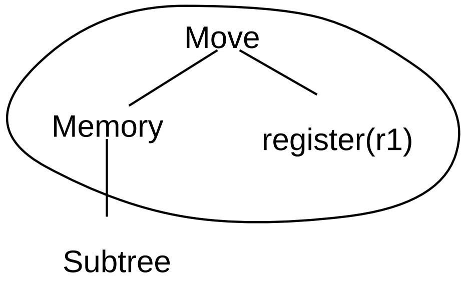
FR-232: **Larger Tiles – Examples**



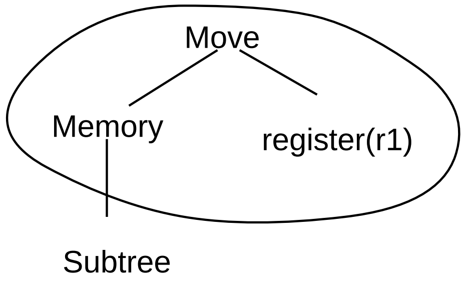
```
<code for Subtree1>
sw   $ACC, 0($ESP)
addi $ESP, $ESP
```

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

FR-233: **Larger Tiles – Examples**

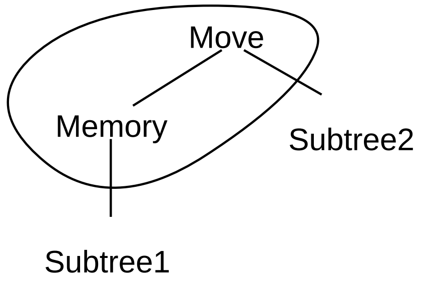


FR-234: **Larger Tiles – Examples**

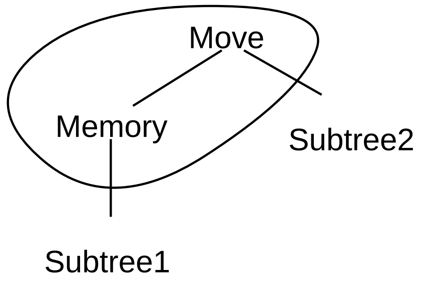


```
<code for Subtree>
sw   $r1, -x($ACC)
```

FR-235: **Larger Tiles – Examples**



FR-236: **Larger Tiles – Examples**



```
<code for Subtree1>
```

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

FR-237: **Optimizing Expression Stack**

- We spend more operations than necessary manipulating the expression stack
- Streamlining stack operations will save us some assembly language instructions (and thus some time)

FR-238: **Constant Stack Offsets**

- Every time we push an item on the Expression Stack, need to increment the $ESP
- Every time we pop an item off the Expression Stack, need to increment the $ESP
- We know at compile time how deep the stack is
  - Can use a constant offset off the $ESP
  - Never need to change the $ESP

FR-239: **Constant Stack Offsets**

- Pushing an expression:

  ```
  sw   $ACC, 0($ESP)
  addi $ESP, $ESP,   -4
  ```

- Popping an expression:

  ```
  addi $ESP, $ESP,   4
  lw   $ACC, 0($ESP)
  ```

FR-240: **Constant Stack Offsets**

- Pushing an expression:

  ```
  sw   $ACC, <offset>($ESP)
  ```

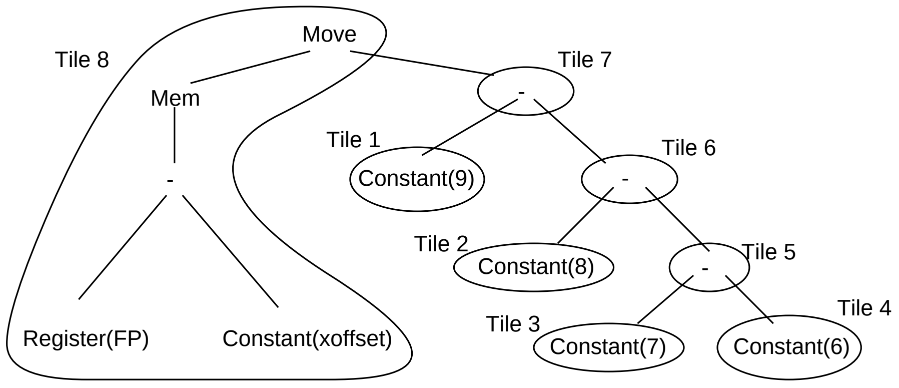
  (decement <offset> by 4)

- Popping an expression:
  (increment <offset> by 4)

  ```
  lw   $ACC, <offset>($ESP)
  ```

FR-241: **Constant Stack Offsets**

- Example:



FR-242: **Constant Stack Offsets**

```
addi $ACC, $zero,  9          % Tile 1
sw   $ACC, 0($ESP)            % Tile 7 -- pushing a value on the
addi $ESP, $ESP,  -4          % Tile 7          expression stack
addi $ACC, $zero,  8          % Tile 2
sw   $ACC, 0($ESP)            % Tile 6 -- pushing a value on the
addi $ESP, $ESP,  -4          % Tile 6          expression stack
addi $ACC, $zero,  7          % Tile 3
sw   $ACC, 0($ESP)            % Tile 5 -- pushing a value on the
addi $ESP, $ESP,  -4          % Tile 5          expression stack
addi $ACC, $zero,  6          % Tile 4
addi $ESP, $ESP,   4          % Tile 5 -- popping a value off the
lw   $t1,  0($ESP)            % Tile 5          expression stack
sub  $ACC, $t1, $ACC          % Tile 5
addi $ESP, $ESP,   4          % Tile 6 -- popping a value off the
lw   $t1,  0($ESP)            % Tile 6          expression stack
sub  $ACC, $t1, $ACC          % Tile 6
addi $ESP, $ESP,   4          % Tile 7 -- popping a value off the
lw   $t1,  0($ESP)            % Tile 7          expression stack
sub  $ACC, $t1, $ACC          % Tile 7
lw   $ACC, xoffset($FP)       % Tile 8
```

FR-243: **Constant Stack Offsets**

```
addi $ACC, $zero,  9          % Tile 1
sw   $ACC, 0($ESP)            % Tile 7 -- pushing on expression stack
addi $ACC, $zero,  8          % Tile 2
sw   $ACC, -4($ESP)           % Tile 6 -- pushing on expression stack
addi $ACC, $zero,  7          % Tile 3
sw   $ACC, -8($ESP)           % Tile 5 -- pushing on expression stack
addi $ACC, $zero,  6          % Tile 4
lw   $t1,  -8($ESP)           % Tile 5 -- popping from expresssion stack
sub  $ACC, $t1, $ACC          % Tile 5
lw   $t1,  -4($ESP)           % Tile 6 -- popping from expresssion stack
sub  $ACC, $t1, $ACC          % Tile 6
lw   $t1,  0($ESP)            % Tile 7 -- popping from expresssion stack
sub  $ACC, $t1, $ACC          % Tile 7
lw   $ACC, xoffset($FP)       % Tile 8
```

FR-244: **Constant Stack Offsets**

- Using constant offsets off the $ESP words well most of the time
- There are problems with function calls, however
  - `x = y + (z + bar(2))`

FR-245: **Constant Stack Offsets**

- `x = y + (z + bar(2))`

```
lw   $ACC, -8($FP)       % Load y into the ACC
sw   $ACC, 0($ESP)       % Store value of y on the expression stack
lw   $ACC, -12($FP)      % Load z into the ACC
sw   $ACC, -4($ESP)      % Store value of z on the expression stack
addi $ACC, $zero,   2    % Load actual parameter (2) into ACC
sw   $ACC, 0($SP)        % Store actual parameter on the stack
addi $SP,  $SP,    -4    % Adjust the stack pointer
jal  bar                 % Make the function call
addi $SP,  $SP,     4    % Adjust the stack pointer
addi $ACC, $result, 0    % Store result of function call in the ACC
lw   $t1,  -4($ESP)      % Load stored value of z into t1
add  $ACC, $t1, $ACC     % Do the addtion z + bar(2)
lw   $t1,  0($ESP)       % Load stored value of y into t1
addi $ACC, $t1, $ACC     % Do the addition y + (z + bar(2))
sw   $ACC, -4($FP)       % Store value of addition in x
```

- What's wrong with this code?

FR-246: **Constant Stack Offsets**

- What's wrong with this code?
  - When we call the function, constant offset is -8.
    - There are 2 expressions stored *beyond* the top of the $ESP
  - In the body of the function, constant offset is ..

FR-247: **Constant Stack Offsets**

- What's wrong with this code?
  - When we call the function, constant offset is -8.
    - There are 2 expressions stored *beyond* the top of the $ESP
  - In the body of the function, constant offset is 0!
  - If `bar` uses the expression stack, it will clobber the values we've stored on it!

FR-248: **Constant Stack Offsets**

- Problem:
  - Function calls expect constant offset to be 0 at start of the function
  - Actual constant offset may not be 0
    - May be *arbitrarily large* (why?)
- Solution:

FR-249: **Constant Stack Offsets**

- Problem:
  - Function calls expect constant offset to be 0 at start of the function
  - Actual constant offset may not be 0
    - May be *arbitrarily large* (why?)
- Solution:
  - Before a function call, decrement the $ESP by constant offset
    - Constant offset is now 0 again
  - After the function call, increment the $ESP again

FR-250: **Constant Stack Offsets**

- `x = y + (z + bar(2))` (Corrected)

```
lw   $ACC, -8($FP)      % Load y into the ACC
sw   $ACC, 0($ESP)      % Store value of y on the expression stack
lw   $ACC, -12($FP)     % Load z into the ACC
sw   $ACC, -4($ESP)     % Store value of z on the expression stack
addi $ACC, $zero,   2   % Load actual parameter (2) into ACC
sw   $ACC, 0($SP)       % Store actual parameter on the stack
addi $SP,  $SP,    -4   % Adjust the stack pointer
addi $ESP, $ESP, -8     % ** Adjust the expression stack pointer **
jal  bar                % Make the function call
addi $ESP, $ESP, 8      % ** Adjust the expression stack pointer **
addi $SP,  $SP,     4   % Adjust the stack pointer
addi $ACC, $result, 0   % Store result of function call in the ACC
lw   $t1,  -4($ESP)     % Load stored value of z into t1
add  $ACC, $t1, $ACC    % Do the addtion z + bar(2)
lw   $t1,  0($ESP)      % Load stored value of y into t1
addi $ACC, $t1, $ACC    % Do the addition y + (z + bar(2))
sw   $ACC, -4($FP)      % Store value of addition in x
```

FR-251: **Optimizing Expression Stack**

- Like to store temporary values in Registers instead of in main memory
- Can't store *all* temporary values in Registers
  - Arbitrarily large number of temporary values may be required
  - Limited number of registers
- Can store *some* temporary values in registers

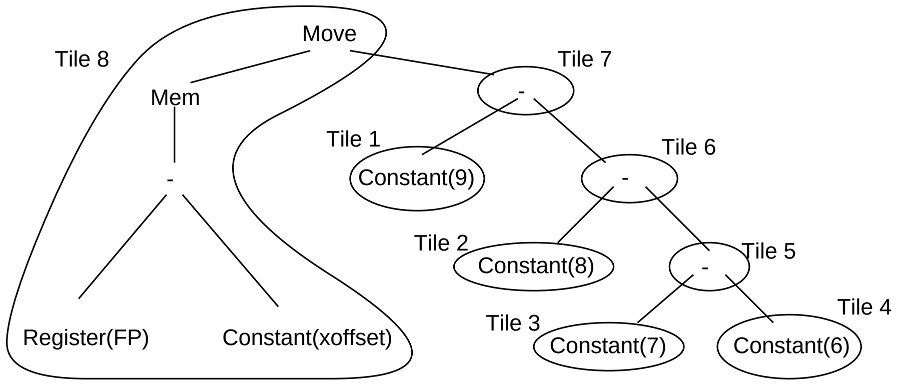
FR-252: **Using Registers**

- Store the bottom of the expression stack in registers
  - For small expressions, we will not need to use main memory for temporary values
  - Retain the flexibility to handle large expressions
- Bottom $x$ elements of the expression stack in registers

FR-253: **Using Registers**

- Example:
  - Use two temporary registers $r2 & $r3
  - If we only need two temporary values, use $r2 and $r3
  - When more values are required, use stack

FR-254: **Using Registers**



FR-255: **Using Registers**

- Constant stack offsets (no registers)

```
addi $ACC, $zero,  9        % Tile 1
sw   $ACC, 0($ESP)          % Tile 7 -- pushing on expression stack
addi $ACC, $zero,  8        % Tile 2
sw   $ACC, -4($ESP)         % Tile 6 -- pushing on expression stack
addi $ACC, $zero,  7        % Tile 3
sw   $ACC, -8($ESP)         % Tile 5 -- pushing on expression stack
addi $ACC, $zero,  6        % Tile 4
lw   $t1,  -8($ESP)         % Tile 5 -- popping from expresssion stack
sub  $ACC, $t1, $ACC        % Tile 5
lw   $t1,  -4($ESP)         % Tile 6 -- popping from expresssion stack
sub  $ACC, $t1, $ACC        % Tile 6
lw   $t1,  0($ESP)          % Tile 7 -- popping from expresssion stack
sub  $ACC, $t1, $ACC        % Tile 7
lw   $ACC, xoffset($FP)     % Tile 8
```

FR-256: **Using Registers**

- Bottom of expression stack in registers

```
addi $ACC, $zero,  9         % Tile 1
addi $r2,  $ACC,   0         % Tile 7 -- pushing on expression stack
addi $ACC, $zero,  8         % Tile 2
addi $r3,  $ACC,   0         % Tile 6 -- pushing on expression stack
addi $ACC, $zero,  7         % Tile 3
sw   $ACC, 0($ESP)           % Tile 5 -- pushing on expression stack
addi $ACC, $zero,  6         % Tile 4
lw   $t1,  0($ESP)           % Tile 5 -- popping from expresssion stack
sub  $ACC, $t1, $ACC         % Tile 5
addi $t1   $t3,    0         % Tile 6 -- popping from expresssion stack
sub  $ACC, $t1, $ACC         % Tile 6
addi $t1,  $t2,    0         % Tile 7 -- popping from expresssion stack
sub  $ACC, $t1, $ACC         % Tile 7
lw   $ACC, xoffset($FP)      % Tile 8
```

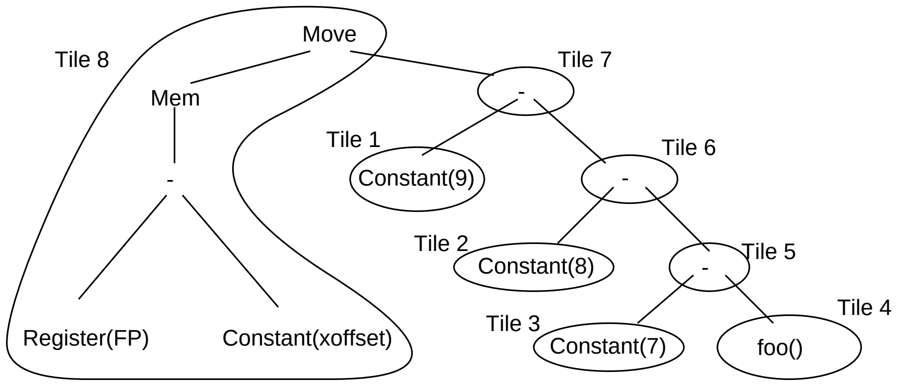
FR-257: **Using Registers**

- If we store the bootom of the expression stack in regisers, we have a problem with function calls:
  - x = foo(a,b) + foo(c,d)
- What can we do?

FR-258: **Using Registers**

- If we store the bootom of the expression stack in regisers, we have a problem with function calls:
  - x = foo(a,b) + foo(c,d)
- What can we do?
  - On a function call, push all registers onto the expression stack, and update the expression stack pointer.
  - After a function call, pop all values back into the registers, and update the expression stack pointer

FR-259: **Using Registers**



FR-260: **Using Registers**

```
addi $ACC, $zero,  9         % Tile 1
addi $r2,  $ACC,   0         % Tile 7 -- pushing on expression stack
addi $ACC, $zero,  8         % Tile 2
addi $r3,  $ACC,   0         % Tile 6 -- pushing on expression stack
addi $ACC, $zero,  7         % Tile 3
sw   $ACC, 0($ESP)           % Tile 5 -- pushing on expression stack
sw   $r2, -4($ESP)           % Tile 4 -- Push register on ESP
sw   $r3, -8($ESP)           % Tile 4 -- Push register on ESP
addi $ESP, $ESP, -12         % Tile 4 -- Update ESP
jal  foo                     % Tile 4
addi $ACC, $result, 0        % Tile 4
addi $ESP, $ESP, 12          % Tile 4 -- Update ESP
lw   $r2,  -4($ESP)          % Tile 4 -- Pop register off ESP
lw   $r3,  -8($ESP)          % Tile 4 -- Pop register off ESP
lw   $t1,  0($ESP)           % Tile 5 -- popping from expresssion stack
sub  $ACC, $t1, $ACC         % Tile 5
addi $t1   $t3,    0         % Tile 6 -- popping from expresssion stack
sub  $ACC, $t1, $ACC         % Tile 6
addi $t1,  $t2,    0         % Tile 7 -- popping from expresssion stack
sub  $ACC, $t1, $ACC         % Tile 7
lw   $ACC, xoffset($FP)      % Tile 8
```

FR-261: **Extended Example**

```
if (x > 1)
   x = y;
else
   x = z + 1;
```

- x has offset 4
- y has offset 8
- z has offset 12

FR-262: **Registers vs. Memory**

- Large quantity of registers (32) at our disposal
- Registers are much faster to access than main memory
- Currently generated code does not use registers efficiently
    - Variables are stored on the stack (main memory), instead of in registers

FR-263: **Registers vs. Memory**

- Since registers are so much faster than main memory, use them as much as possible
- Two reasons why we can't use only registers for variables
    - Escaping Variables
    - Spilled Registers

FR-264: **Escaping Variables**

- A variable *escapes* if we need to store it in memory
    - Calculate the address of a variable (implicitly or explicitly)
    - Use the address of the variable to access it
- When do we need to calculate the address of a variable?

FR-265: **Escaping Variables**

- When do we need to calculate the address of a variable?
    - Explicitly use the & operator in C
    - Pass-by-reference (C++)
    - Arrays / strings
    - Global / extern variables

FR-266: **Spilling Registers**

- Only have a finite number of registers (32 in MIPS)
- Programs can require an arbitrarily large number of variables

- Have more variables to store in registers than registers to use
    - Register *spill* into memory

FR-267: **Register Allocation**

- How can we determine which variables are stored in which registers, and which values need to spill into memory?
- How can we determine which variables to store in which registers?
- Register assignment can be non-trivial

FR-268: **Register Allocation**

```
a = 3           addi a, $zero, 3
c = a + 4       addi c, a, 4
b = 2           addi b, $zero, 2
c = b + c       add  c, b, c
a = c + 2       addi a, c, 2
c = a + 1       addi c, a, i
```

FR-269: **Register Allocation**

```
a = 3           addi a, $zero, 3
c = a + 4       addi c, a, 4
b = 2           addi b, $zero, 2
c = b + c       add  c, b, c
a = c + 2       addi a, c, 2
c = a + 1       addi c, a, i
```

- Code has 3 variables
- Could implement it with only 2 registers
- Never need to keep track of value of `a` and `b` at the same time

FR-270: **Interference**

- Need to maintain the values of two variables `x` and `y` simultaneously
- `x` and `y` *interfere* with each other
- Can't store `x` and `y` in the same register

FR-271: **Register Allocation**

- Assume that we have an infinite number of registers
- Every time we declare a new variable, create a new abstract register
- After we have created assembly, replace abstract registers with actual registers
    - Use registers as efficiently as possible
    - May have to spill some of the abstract registers into memory

FR-272: **Register Allocation**

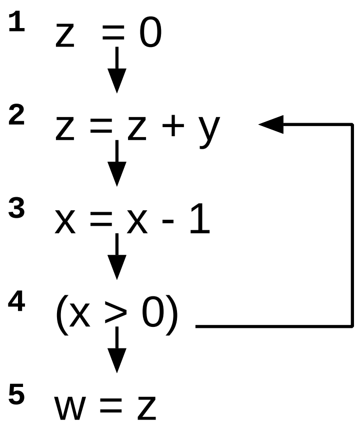
- Create assembly that uses abstract registers
- Figure our which abstract registers interfere with each other
- Use this information to assign actual registers to abstract registers as efficiently as possible

FR-273: **Control Flow Graph**

- A Control Flow Graph describes the flow of control through an assembly language program
- Nodes in the graph are assembly language statements
- Edge from node A to node B if statement in node B can immediately follow the statement in node A

FR-274: **Control Flow Graph**

```
    z = 0
L1: z = z + y
    x = x - 1
    if (x>0) goto L1
    w = z
```



FR-275: **Definitions**

- Node p is a *successor* of Node n if there is an arc from Node n to Node p
- successors[n] = set of all successors of n
- Node p is a *predecessor* of Node n if there is an arc from Node p to Node n
- predecessors[n] = set of all predecessors of n

FR-276: **Liveness Analysis**

- We will use the Control Flow Graph to do *Liveness Analysis*
- A variable is "live" on an edge in the control flow graph if changing the value of the variable at that point in the execution will alter the behavior of the program
- Variable is live on an edge if we care about the value of the variable at that point in the program

FR-277: **Liveness Analysis**

- "Live In"
  - Variable is live-in to a node if it is live on any edge entering the node

- live-in[n] = set of all variables live-in to n

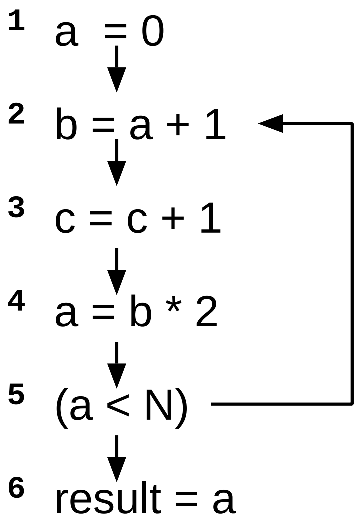
- "Live Out"
  - Variable is live-out from a node if it is live on any edge leaving the node
  - live-out[n] = set of all variables live-out from n

FR-278: **Liveness Analysis**

- To calculate live-in[n] and live-out[n], we will need to know which variables are *used* at each node, and which variables are *defined* at each node
  - use[n] = set of all variables whose values are used at node n
  - define[n] = set of all variables whose values are defined (set) at node n

FR-279: **Control Flow Graph**

```
    a = 0
L1:
    b = a + 1
    c = c + 1
    a = b * 2
    if  (a < N) goto L1
    return a
```



FR-280: **Use / Define**

| Node | Use | Define |
|---|---|---|
| 1 | { } | {a} |
| 2 | {a } | {b} |
| 3 | {c } | {c} |
| 4 | {b } | {a} |
| 5 | {a,N } | { } |
| 6 | {a } | {result} |

FR-281: **Live-in / Live-out**

- live-in and live-out can be calculated as follows:

$$
\begin{aligned}
out[n] &= \bigcup_{s \in successors[n]} in[s] \\
in[n] &= use[n] \bigcup (out[n] - define[n])
\end{aligned}
$$

FR-282: **Live-in / Live-out**

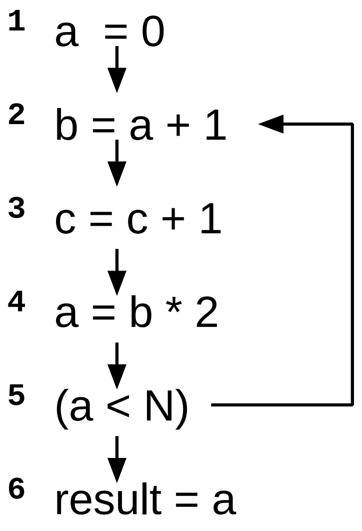
- To calculate live-in and live-out for all nodes:
  - Set in[n] = {}, out[n] = {} for all n
  - For each node n, recalculate in[n] and out[n] using

$$\begin{aligned} out[n] &= \bigcup_{s \in successors[n]} in[s] \\ in[n] &= use[n] \bigcup (out[n] - define[n]) \end{aligned}$$

  - Repeat until no changes are made to in[], out[]

FR-283: **Liveness Analysis Example**

```
   a = 0
L1:
   b = a + 1
   c = c + 1
   a = b * 2
   if  (a < N) goto L1
   return a
```



FR-284: **Liveness Analysis Example**

| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | { } | {a } | {N, c} | {a, c, N } |
| 2 | {a} | {b } | {a, c, N} | {b, c, N} |
| 3 | {c} | {c } | {b, c, N} | {b, c, N} |
| 4 | {b} | {a } | {b, c, N} | {a, c, N} |
| 5 | {a, N} | { } | {a, c, N } | {a, c, N} |
| 6 | {a } | {result } | {a } | { } |

FR-285: **Interference Graphs**

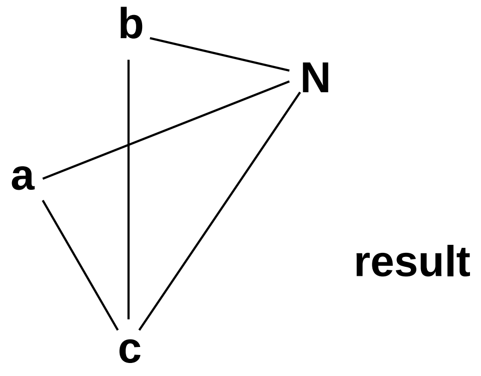
- We will use interference graphs to assist in register allocation
- Each node in the graph is a variable
- Edge between two variables if they interfere with each other
  - Need to be in different registers
  - Live at the same time

FR-286: **Interference Graphs**

- For each node n in the control flow graph

- For each variable x ∈ define[n]
  - For each variable y ∈ out[n], add an edge between x and y

FR-287: **Interference Graphs**



FR-288: **Register Allocation**

- Once we have the interference graph, register allocation is just graph coloring
  - Assign colors to all vertices so that adjacent vertices have different colors
  - Colors – actual registers

FR-289: **Register Allocation**

- Once we have the interference graph, register allocation is just graph coloring
  - Assign colors to all vertices so that adjacent vertices have different colors
  - Colors – actual registers
- Graph coloring is NP-Complete
  - Use an approximate solution
  - Won't always guarantee the smallest number of colors are used
  - Run in polynomial time

FR-290: **Graph Coloring**

- Don't need to find an optimal coloring
- Need to find a coloring that uses fewer than $k$ colors
  - $k$ is the number of actual registers
- Won't always be able to find a $k$-coloring when one exists (NP-Complete), but will find a $k$-coloring *most* of the time

FR-291: **Significant Degree**

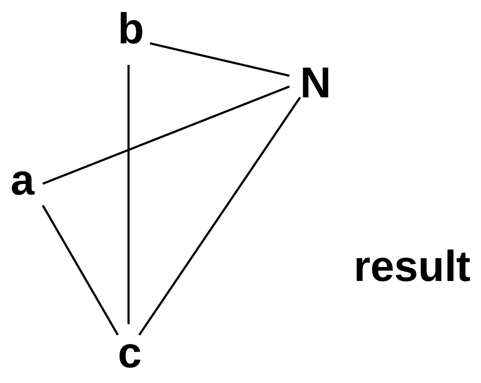
- "Degree" of a vertex is the number of neighbors that it has
- Vertex with $\geq k$ neighbors is of "Significant Degree"

- If a vertex is *not* of significant degree, then it can always be colored – *regardless of the colors chosen for the rest of the graph*
  - Fewer than $k$ neighbors
  - $k$ colors – can always choose a color different from all neighbors

FR-292: **Graph Coloring**

1. Initialize color stack to be empty
2. Select a node $x$ that is not of significant degree. Push $x$ on color stack, remove $x$ from the graph
3. If there are any nodes left, go to step 2
4. While the color stack is not empty
   (a) Pop the top variable off the color stack
   (b) Color this variable with an arbitrary color, different from already colored neighbors

FR-293: **Graph Coloring Example**



- 3-Color this graph

FR-294: **Register Allocation**

$$\begin{tikzcd}
\text{Liveness Analysis} \arrow[d] \\
\text{Build Interference Graph} \arrow[d] \\
\text{Simplify} \arrow[loop right] \arrow[d] \\
\text{Color}
\end{tikzcd}$$

FR-295: **Spilling**

- Not every graph can be $k$-colored, for all $k$
- Previous graph cannot be 2-colored
- If we don't have enough registers, some values need to "spill" into memory

FR-296: **Spilling**

1. Initialize the color stack to be empty
2. If there are any variables x that are not of significant degree
    - Remove x from the interference graph, and push x onto the color stack.

   else

    - Pick a node y to potentially store in memory. Push y on the color stack, and remove y from the interference graph
3. If there are any nodes left, go to step 2

FR-297: **Spilling**

4. Restore the interference graph
5. While the color stack is not empty:
    (a) Pop the top variable off the color stack
    (b) If there is an available color for x that is different than the colors already chosen for x's neighbors:
        - Color x

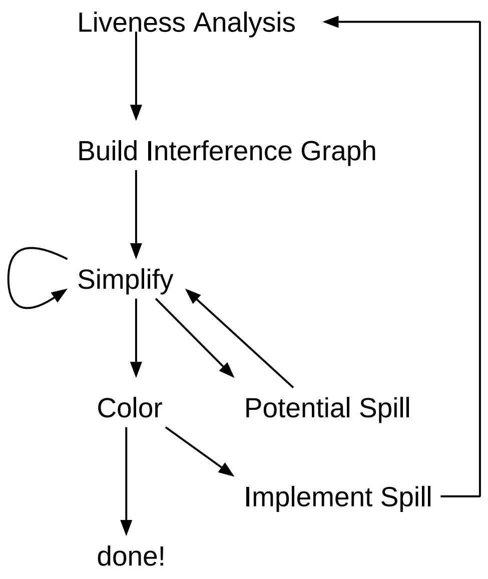
        else

        - Mark x as an actual spill
6. If no variables spilled, we're done. If at least one variable spilled, implement the spill.

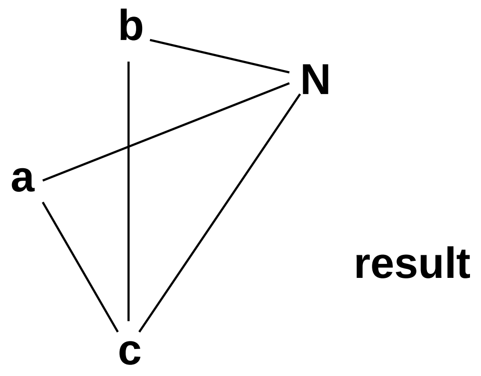
FR-298: **Spilling**

- If a register *spills*, it must be stored in memory
    - Before every use, read variable from memory
    - After every use, write variable back to memory
    - "Live" period of register will be very short
    - Should be able to color the graph

FR-299: **Register Allocation**

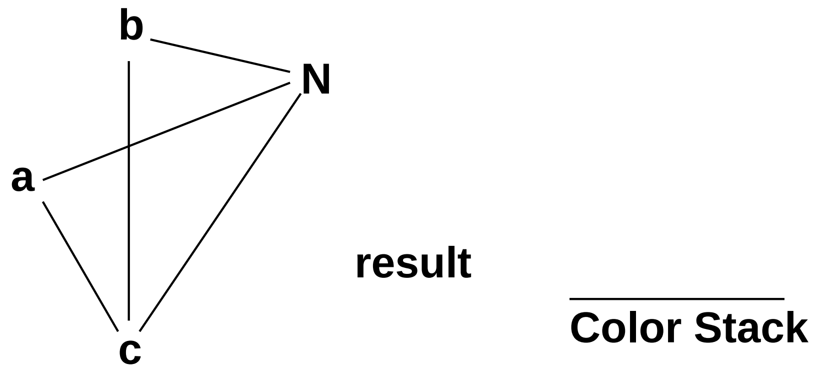


FR-300: **Register Allocation**



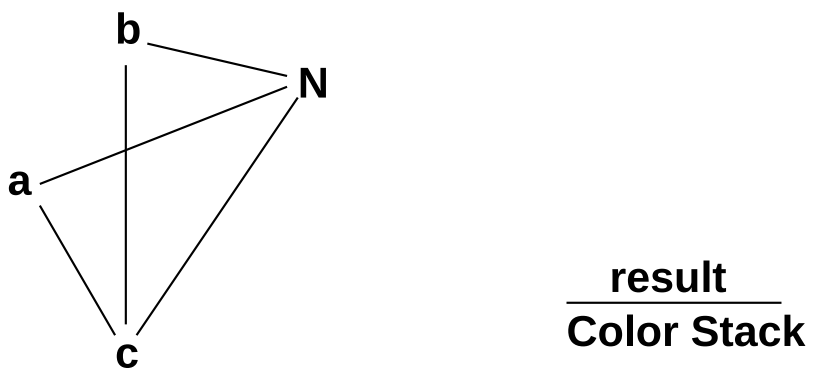
- Try to 2-color the interference graph

FR-301: **Register Allocation**



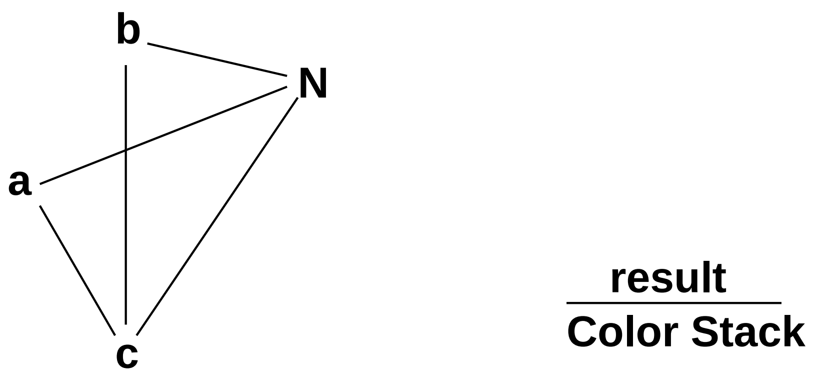
- Try to 2-color the interference graph

FR-302: **Register Allocation**



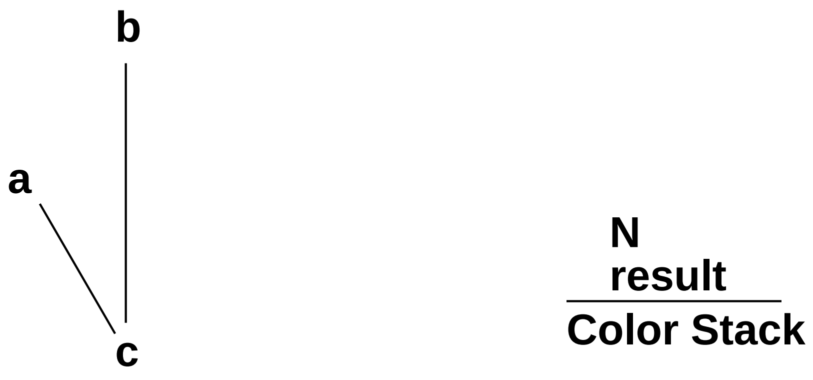
- All vertices are of significant degree
- Pick one to (potentially) store in memory
  - Which one?

FR-303: **Register Allocation**



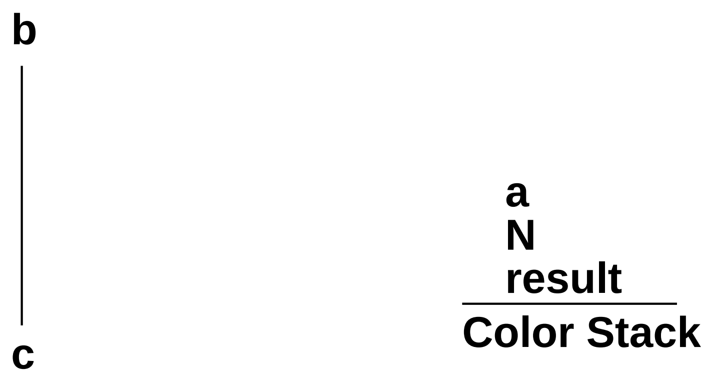
- All vertices are of significant degree
- Pick one to (potentially) store in memory
  - N interferes with many other variables
  - Used infrequently

FR-304: **Register Allocation**



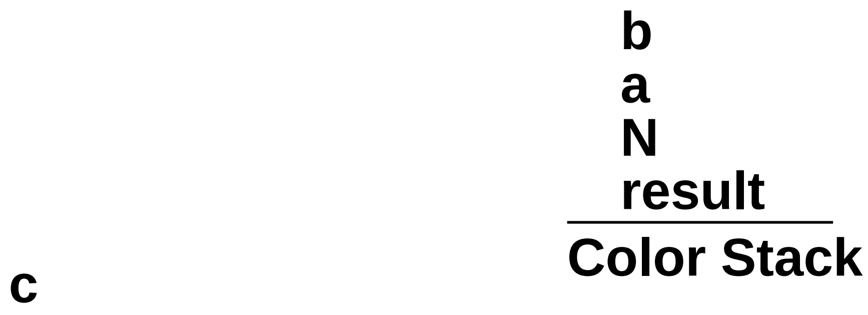
- Remove N
    - Potential Spill

FR-305: **Register Allocation**



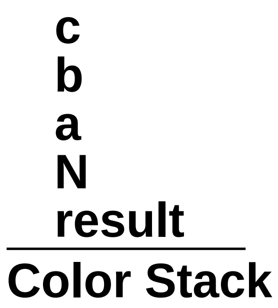
- Continue removing vertices of non-significant degree

FR-306: **Register Allocation**



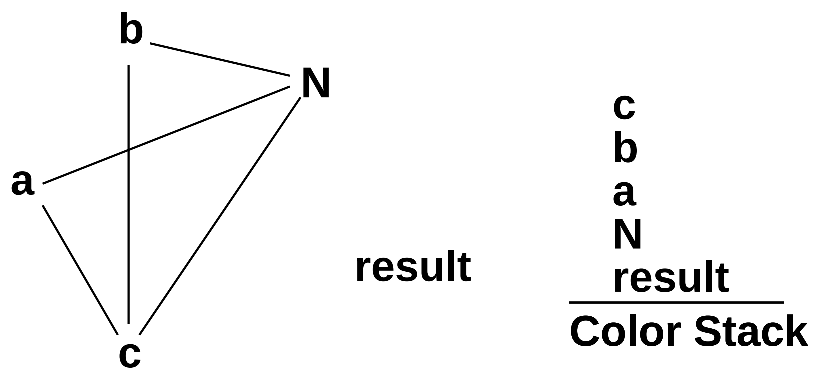
- Continue removing vertices of non-significant degree

FR-307: **Register Allocation**



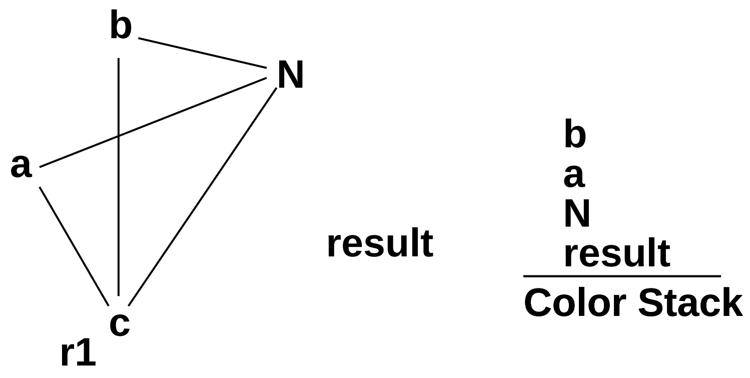
- Continue removing vertices of non-significant degree

FR-308: **Register Allocation**



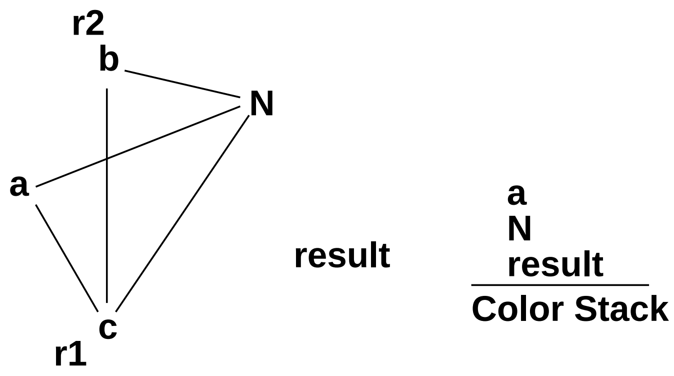
- Pop variables off color stack, and color graph

FR-309: **Register Allocation**



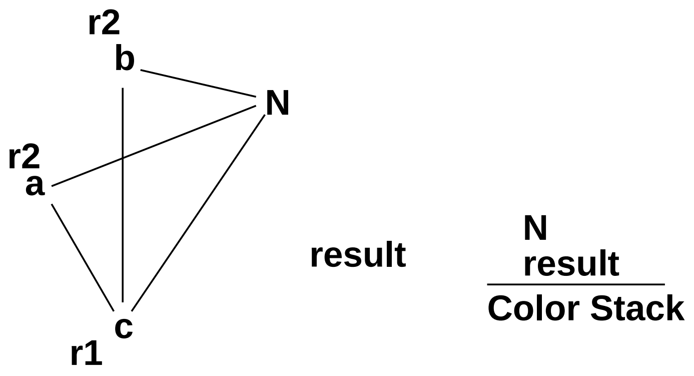
- Pop variables off color stack, and color graph

FR-310: **Register Allocation**



- Pop variables off color stack, and color graph

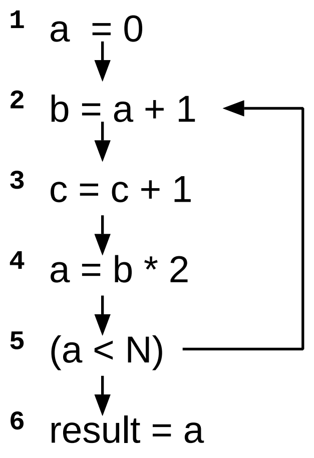
FR-311: **Register Allocation**



- Can’t color N
  - N is an actual spill

FR-312: **Register Allocation**

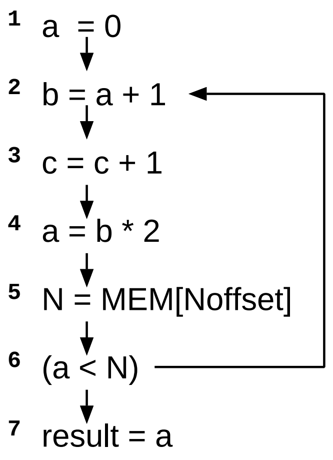
```
    a = 0
L1:
    b = a + 1
    c = c + 1
    a = b * 2
    if  (a < N) goto L1
    return a
```



- Before every use of N, load from memory
- After every set of N (none in this case), store back into memory

FR-313: **Register Allocation**

```
    a = 0
L1:
    b = a + 1
    c = c + 1
    a = b * 2
    N = MEM[Noffset]
    if  (a < N) goto L1
    return a
```

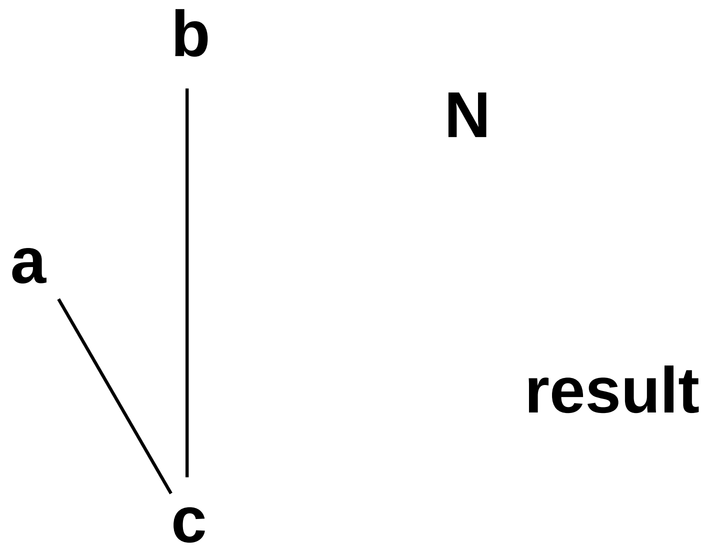


- Before every use of N, load from memory
- After every set of N (none in this case), store back into memory

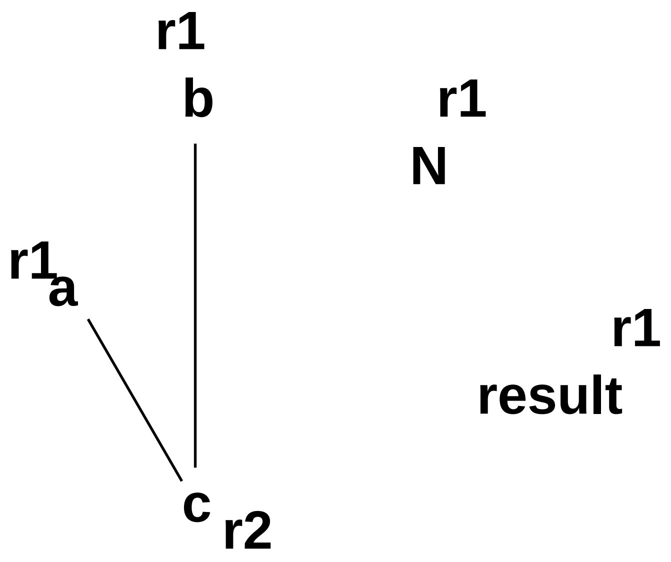
FR-314: **Register Allocation**

| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | { } | {a } | {c} | {a, c, } |
| 2 | {a} | {b } | {a, c, } | {b, c, } |
| 3 | {c} | {c } | {b, c, } | {b, c, } |
| 4 | {b} | {a } | {b, c, } | {a, c, } |
| 5 | {} | { N } | {a, c, } | {a, c, N} |
| 6 | {a, N} | { } | {a, c, N } | {a, c } |
| 7 | {a } | {result } | {a } | { } |

FR-315: **Register Allocation**



FR-316: **Register Allocation**



FR-317: **Register Allocation**

```
d = 2                   addi d, zero, 2
b = 3                   addi b, zero, 3
a = 4                   addi a, zero 4
a = a + 1               addi a, a, 1
c = 4                   addi c, zero, 4
b = b + c               add  b, b, c
d = d + 1               addi d, d, 1
```

FR-318: **Register Allocation**

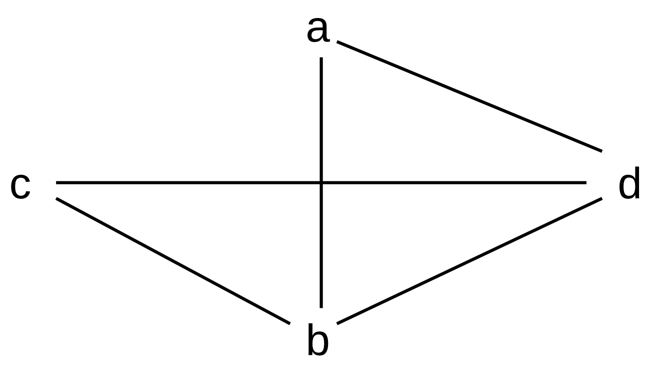
d = 2
b = 3
a = 4
a = a + 1
c = 4
b = b + c
d = d + 1

$$\begin{tikzcd}
\textbf{1}\ d = 2 \arrow[d] \\
\textbf{2}\ b = 3 \arrow[d] \\
\textbf{3}\ a = 4 \arrow[d] \\
\textbf{4}\ a = a + 1 \arrow[d] \\
\textbf{5}\ c = 4 \arrow[d] \\
\textbf{6}\ b = b + c \arrow[d] \\
\textbf{7}\ d = d + 1
\end{tikzcd}$$

FR-319: **Register Allocation**

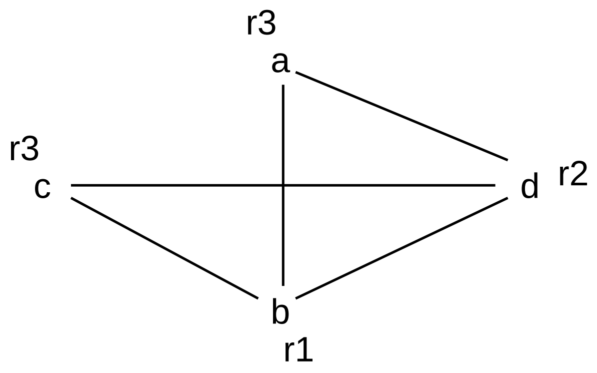
| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | { } | {d } | { } | {d } |
| 2 | { } | {b } | {d} | {b, d} |
| 3 | { } | {a } | {b, d} | {b, a, d} |
| 4 | {a} | {a } | {b, a, d} | {b, d} |
| 5 | {} | {c } | {b, d} | {b, c, d} |
| 6 | {b, c} | {b } | {b, c, d} | {d} |
| 7 | {d } | {d } | {d} | { } |

FR-320: **Register Allocation**



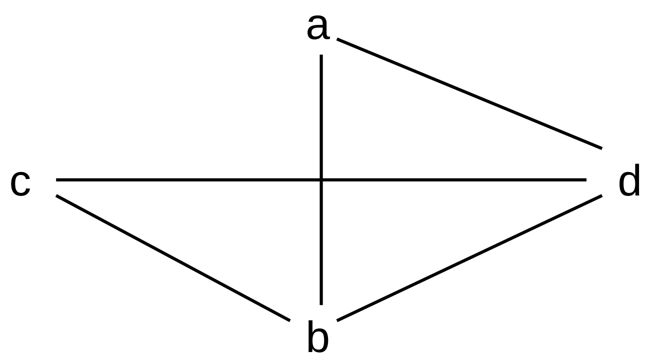
- 3-color

FR-321: **Register Allocation**



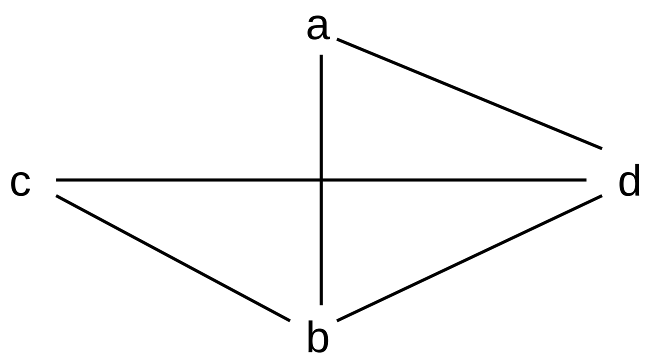
- 3-color

FR-322: **Register Allocation**



- 2-color

FR-323: **Register Allocation**



- 2-color
  - What should we pick as a potential spill?

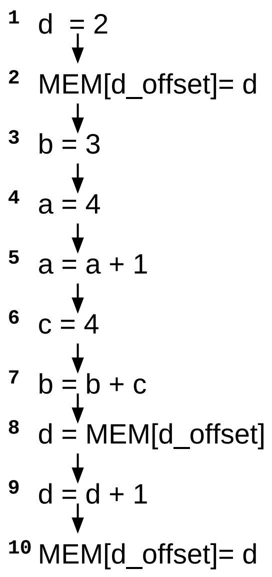
FR-324: **Register Allocation**

```
d = 2               addi d, zero, 2
b = 3               addi b, zero, 3
a = 4               addi a, zero 4
a = a + 1           addi a, a, 1
c = 4               addi c, zero, 4
b = b + c           add  b, b, c
d = d + 1           addi d, d, 1
```

FR-325: **Register Allocation**

```
d = 2               addi d, zero, 2
MEM[d_offset] = d   sw   d, (d_offset) fp
b = 3               addi b, zero, 3
a = 4               addi a, zero 4
a = a + 1           addi a, a, 1
c = 4               addi c, zero, 4
b = b + c           add  b, b, c
d = MEM[d_offset]   lw   d, (d_offset) fp
d = d + 1           addi d, d, 1
MEM[d_offset] = d   sw   d, (d_offset) fp
```

d = 2
MEM[d_offset] = d
b = 3
a = 4
a = a + 1
c = 4
b = b + c
d = MEM[d_offset]
d = d + 1
MEM[d_offset] = d

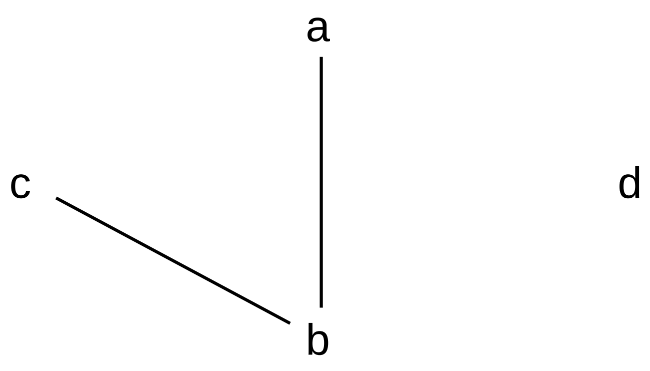


FR-326: **Register Allocation**

FR-327: **Register Allocation**

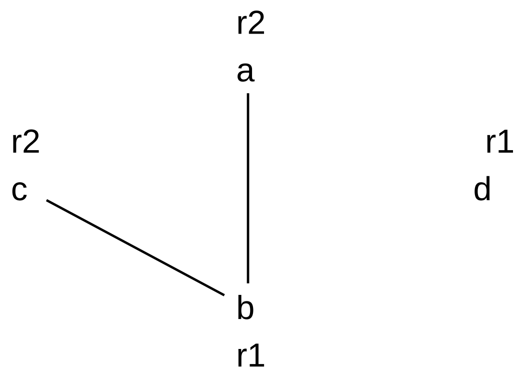
| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | { } | {d} | { } | {d} |
| 2 | {d} | { } | {d} | {} |
| 3 | { } | {b} | {} | {b} |
| 4 | { } | {a} | {b} | {a, b} |
| 5 | {a} | {a} | {a, b} | {b} |
| 6 | { } | {c} | {b} | {b, c} |
| 7 | {b, c} | {b} | {b, c} | { } |
| 8 | { } | {d} | { } | {d} |
| 9 | {d} | {d} | {d} | {d} |
| 10 | {d} | { } | {d} | { } |

FR-328: **Register Allocation**



- 2-color

FR-329: **Register Allocation**



- 2-color

FR-330: **Eliminating Moves**

- When two variables share the same value, we can use the same register for both of them
- Any move from one variable to the other can then be removed.

FR-331: **Eliminating Moves**

```
a = 4
c = a + 1
b = c + 1
d = a
c = d + 1
b = d * c
```

- We can store `a` and `d` in the same register
- The assignment `d = a` is then a no-op, which can be removed

FR-332: **Eliminating Moves**

- Any time there is a move instruction
  - `x = y              (addi x, y, 0)`
- `x` and `y` do *not* interfere (unless they interfere elsewhere)
- If we place `x` and `y` in the same register, we can eliminate the move instruction

FR-333: **Interference Graph, Part II**

- Interference Graph will have two kinds of edges
  - Interference edges
    - Interference edge between `x` and `y`
    - `x` and `y` must go in different registers
  - Move edges
    - Move edge between `x` and `y`
    - Like `x` and `y` to go in the same register
    - Eliminate a move instruction, save some cycles

FR-334: **Interference Graph, Part II**

- For node n in the control flow graph
  - If n is a move instruction of the form x = z
    - Add a move arc between x and z
    - For each variable y in out[n] - z, add an undirected interference arc between x and y.

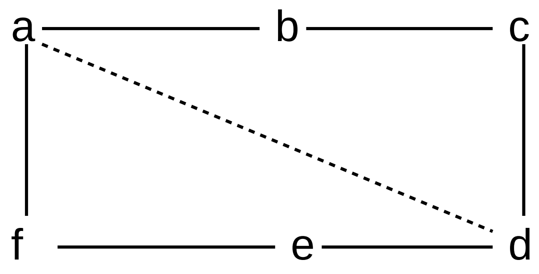
    else if n is not a move instruction
    - For each variable x in define[n] and each variable y in out[n] add an undirected interference arc between x and y
  - Interference Edges trump Move Edges
    - Add both an interference edge and move edge, interference edge wins

FR-335: **Eliminating Moves & Spilling**

- Eliminating moves adds some efficiency
- Reducing spills is more important

    - Cost of going out to memory much higher than cost of an `addi`
- Careful not to be too aggressive in trying to eliminate moves

FR-336: **Eliminating Moves & Spilling**



- Can 2-color this graph
- Cannot 2-color this graph if `a` and `d` have the same color

FR-337: **Register Allocation**

1. Do dataflow analysis to calculate live-out for all nodes in the control-flow graph
2. Create the interference graph
3. Combine all pairs of nodes that are connected by a move arc that will not result in a node with k or more neighbors of significant degree (Coalesce step)

FR-338: **Register Allocation**

4. Initialize the color stack to be empty
5. If there are any variables x that are not of significant degree
    - Remove x from the interference graph, and push x onto the color stack.

   else
    - Pick a node y to potentially store in memory. Push y on the color stack, and remove y from the interference graph
6. If there are any nodes left, go to step 5

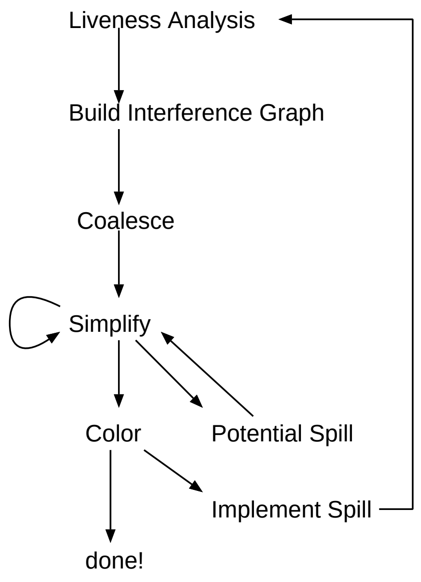
FR-339: **Register Allocation**

7. Restore the interference graph
8. While the color stack is not empty:
    (a) Pop the top variable off the color stack
    (b) If there is an available color for x that is different than the colors already chosen for x's neighbors:
        - Color x

       else
        - Mark x as an actual spill

9. If no variables spilled, we're done. If at least one variable spilled, implement the spill and start over.

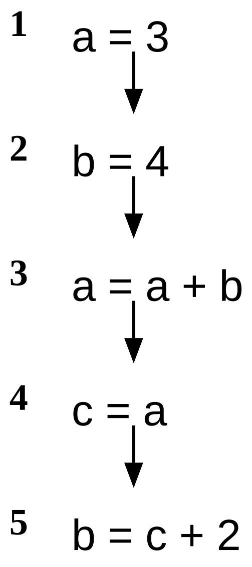
FR-340: **Register Allocation**



FR-341: **Register Allocation**

```
a = 3           addi a, $zero, 3
b = 4           addi b, $zero, 4
a = a + b       add  a, a, b
c = a           addi c, a, 0
b = c + 2       addi b, c, 2
```

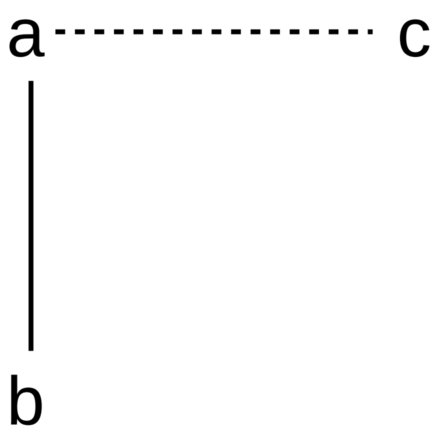
FR-342: **Register Allocation**



FR-343: **Register Allocation**

| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | { } | {a} | { } | {a} |
| 2 | { } | {b} | {a} | {a, b} |
| 3 | {a, b} | {a} | {a, b} | {a} |
| 4 | {a} | {c} | {a} | {c} |
| 5 | {c} | {b} | {c} | { } |

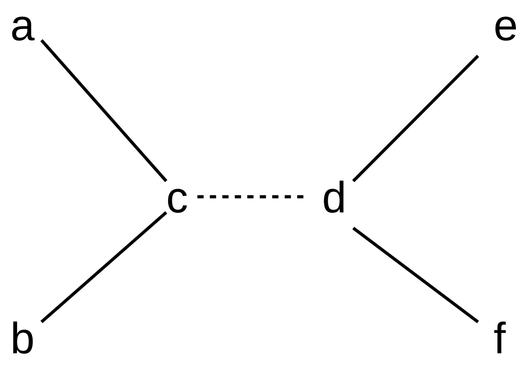
FR-344: **Register Allocation**



FR-345: **Coalescing**

- We need to be sure that no coalesce steps cause unnecessary spills
- If we do all coalescing before any simplifying, we will not always take advantage of move elimination

FR-346: **Coalescing**



- 3-color this graph

FR-347: **Coalescing**

- We need to be sure that no coalesce steps cause unnecessary spills
- If we do all coalescing before any simplifying, we will not always take advantage of move elimination
- Interleave Coalesce and Simplify steps
    - Coalesce as much as we can

- Simplify as much as we can (not simplifying any node that we could possible coalesce later)
- repeat

FR-348: **Register Allocation (Final)**

1. Do dataflow analysis to calculate live-out for all nodes in the control-flow graph
2. Create the interference graph (including move edges)
3. Simplify any variables that are not of significant degree that have no move edges. That is, remove the node from the graph and add it to the color stack.
4. Coalesce (combine) all pairs of nodes that are connected by a move arc that will not result in a node of significant degree.

FR-349: **Register Allocation (Final)**

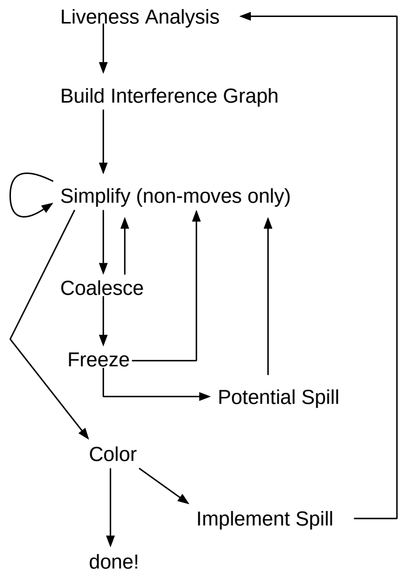
5. If there are any nodes that are not of significant degree that are not involved in any move, go to step 3.
6. If there is a node x that is not of significant degree, which is involved in at least one move, then freeze x – remove all move edges adjacent to x – and goto step 3.
7. If there are any remaining nodes (at this point they all must be of significant degree), pick a node x to potentially spill. Remove x from the graph and place it on the color stack. Goto step 3.

FR-350: **Register Allocation (Final)**

8. At this point, the graph should be empty. Restore the interference graph
9. While the color stack is not empty:
   (a) Pop the top variable off the color stack
   (b) If there is an available color for x that is different than the colors already chosen for x's neighbors:
       - Color x

       else
       - Mark x as an actual spill
10. If no variables spilled, we're done. If at least one variable spilled, implement the spill and start over.

FR-351: **Register Allocation (Final)**

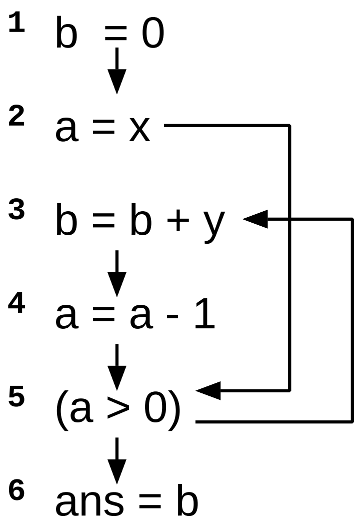


FR-352: **Register Allocation Example**

```
     b = 0                          addi b, zero, 0
     a = x                          addi a, x, 0
     goto test                      b test
body:                          body:
     b = b + y                      add  b, b, y
     a = a - 1                      addi a, a, -1
test:                          test:
     if (a>0) goto body             bgt a
     ans = b                        addi ans, b, 0
```

FR-353: **Register Allocation Example**

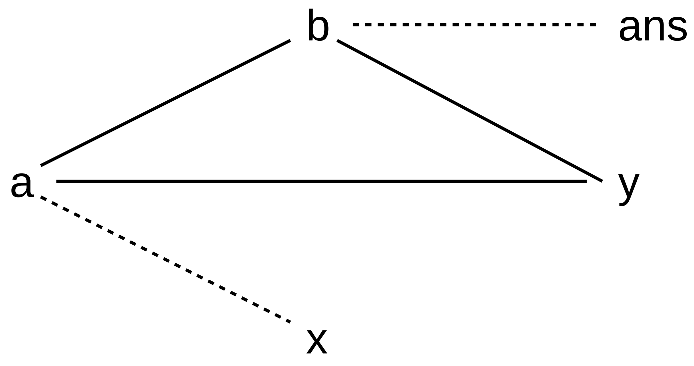
```
     b = 0
      a = x
      goto test
body:
       b = b + y
       a = a - 1
test:
       if (a>0) goto body
       ans = b
```



FR-354: **Register Allocation Example**

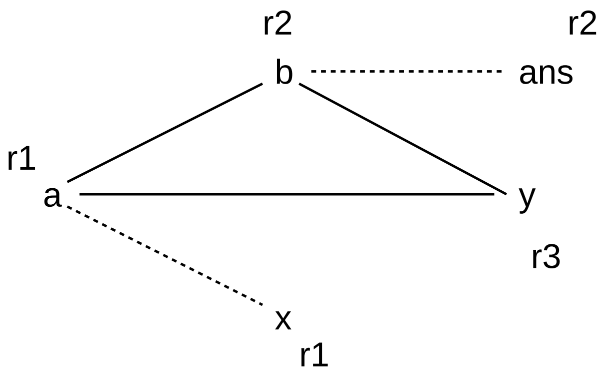
| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | { } | {b} | {x, y} | {b, x, y} |
| 2 | {x} | {a} | {b, x, y} | {a, b, y} |
| 3 | {b, y} | {b} | {a, b, y} | {a, b, y} |
| 4 | {a} | {a} | {a, b, y} | {a, b, y} |
| 5 | {a} | { } | {a, b, y} | {a, b, y} |
| 6 | {b} | {ans} | {b} | {} |

FR-355: **Register Allocation Example**



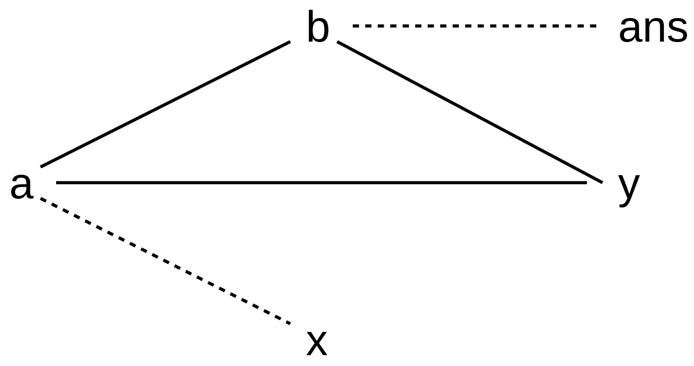
- 3-color graph

FR-356: **Register Allocation Example**



- 3-color graph

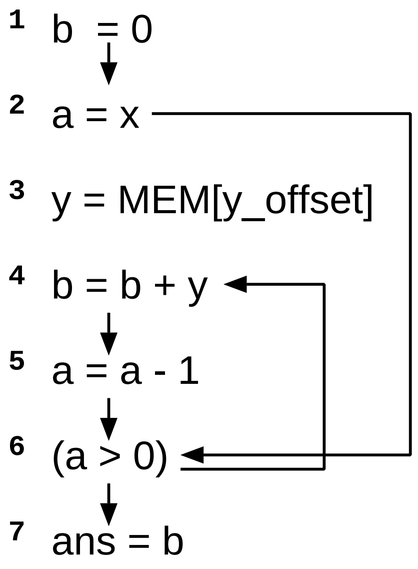
FR-357: **Register Allocation Example**



- 2-color graph

FR-358: **Register Allocation Example**

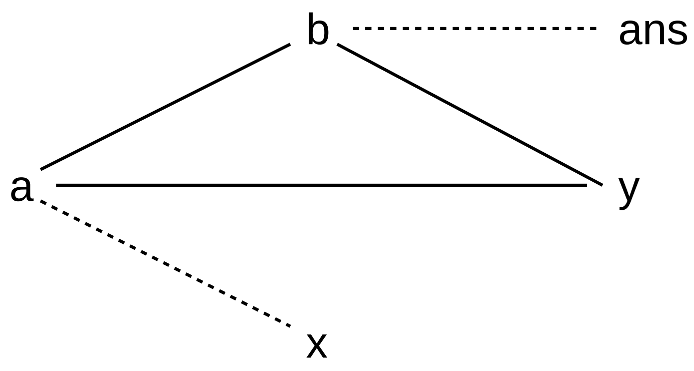
```
      b = 0
       a = x
       goto test
body:
       y=MEM[y_offset]
        b = b + y
        a = a - 1
test:
        if (a>0) goto body
        ans = b
```



FR-359: **Register Allocation Example**

| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | { } | {b} | {x} | {b, x} |
| 2 | {x} | {a} | {b, x} | {a, b} |
| 3 | { } | {y} | {a, b} | {a, b, y} |
| 4 | {b, y} | {b} | {a, b, y} | {a, b} |
| 5 | {a} | {a} | {a, b} | {a, b} |
| 6 | {a} | { } | {a, b} | {a, b} |
| 7 | {b} | {ans} | {b} | {} |

FR-360: **Register Allocation Example**



- Whoops!
- Only 2 registers, need to store everything in memory

FR-361: **Precolored Registgers**

- Some registers (FP, SP, etc) are not variables

- Need to assign specific registers to them
- "Precolor" some of the variables in the graph
    - FP
    - SP
    - ... etc

FR-362: **Parameter Passing**

- Pass parameters on the stack, saving & restoring registers to memory
- Better idea – use registers to pass parameters
    - Identify some registers as "parameter registers"
    - Copy paramters into these registers before function calls
    - Move values out of these registers at start of the function
- Function calls within function calls won't cause problems
- Might be able to eliminate some of the moves

FR-363: **Parameter Passing**

- To call a function, move values of actuals into special (precolored) parameter registers
- At the start of each function, move values of parameters into other variables
    - May be able to color these variables with the same color as precolored – eliminate the move

FR-364: **Parameter Passing**

- Most functions have a (relatively) small number of parameters
- Wasteful to denote too many registers as parameter registers
- Only use 4-5 parameter registers (other parameters are passed on the stack as normal

FR-365: **Caller/Callee Saved**

- One solution to register allocation:
    - Create a control flow graph of the entire program (all function calls, etc)
    - Do dataflow analysis on this graph
    - Allocate registers based on dataflow analysis
- What is wrong with this approach?

FR-366: **Caller/Callee Saved**

- One solution to register allocation:
    - Create a control flow graph of the entire program (all function calls, etc)
    - Do dataflow analysis on this graph
    - Allocate registers based on dataflow analysis

- What is wrong with this approach?
  - Separately compiled functions
  - Libraries
  - etc.

FR-367: **Caller/Callee Saved**

- Do register allocation on a function by function basis
- Denote some registers as "caller-saved"
  - Assume that values of these registers will be destroyed by function calls
  - Save them on the stack if they are live across other function calls
- Denote some registers as "callee-saved"
  - If use these registers, save old values on the stack

FR-368: **Caller/Callee Saved**

- At beginning of a function:
  - Move all callee-saved registers other temporary locations
  - Move all parameters into other locations
- If the function uses any callee saved registers, then the temporary locations will spill into memory – automatically saving registers
- If the function does not need to use any callee saved registers, no unnecessary movement to memory

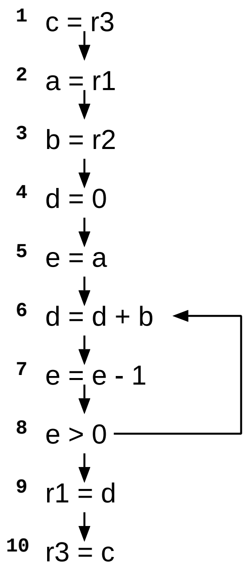
FR-369: **Complete Example**

```
int f(int a, int b) {
   int d = 0
   in e = a;
   do {
     d = d + b;
     e = e - 1;
   while (e > 0);
   return d;
```

- 3 registers, r1, r2, r3
  - r1, r2 are parameter registers
  - r1 is also the result register
  - r3 is callee-saved

FR-370: **Complete Example**

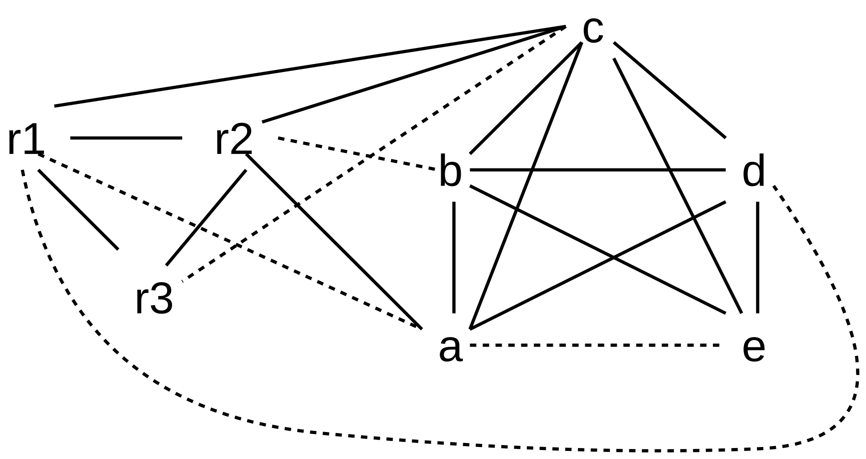
```
      c = r3
      a = r1
      b = r2
      d = 0
      e = a
loop:
      d = d + b
      e = e - 1
     if e > 0 goto loop
     r1 = d
     r3 = c
```



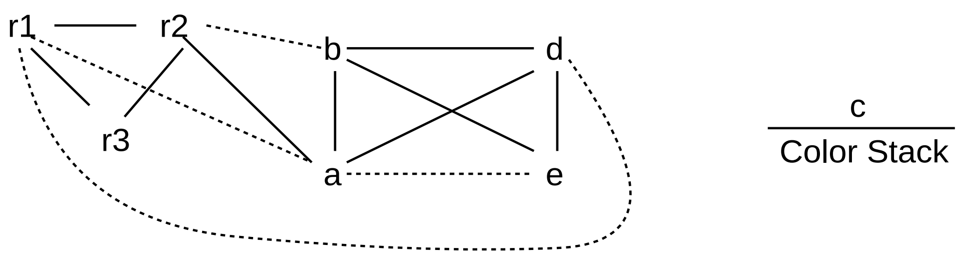
FR-371: **Complete Example**

| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | {r3} | {c} | {r1, r2, r3} | {c, r1, r2} |
| 2 | {r1} | {a} | {c, r1, r2} | {a, c, r2} |
| 3 | {r2} | {b} | {a, c, r2} | {a, b, c} |
| 4 | { } | {d} | {a, b, c} | {a, b, c, d} |
| 5 | {a} | {e} | {a, b, c, d} | {b, c, d, e} |
| 6 | {b, d} | {d} | {b, c, d, e} | {b, c, d, e} |
| 7 | {e} | {e} | {b, c, d, e} | {b, c, d, e} |
| 8 | {e} | { } | {b, c, d, e} | {b, c, d, e} |
| 9 | {d} | {r1} | {c, d} | {r1, c} |
| 10 | {c} | {r3} | {r1, c} | {r1, r3} |

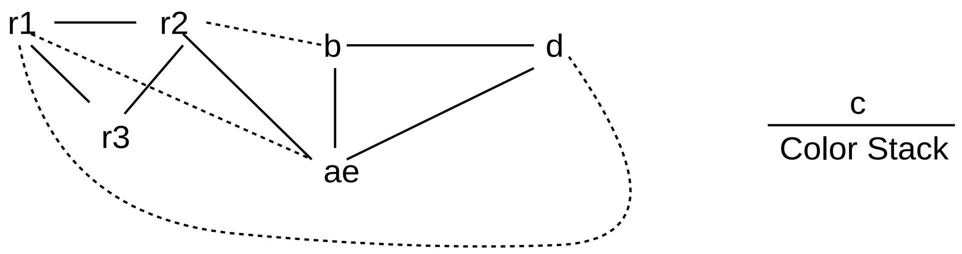
FR-372: **Complete Example**



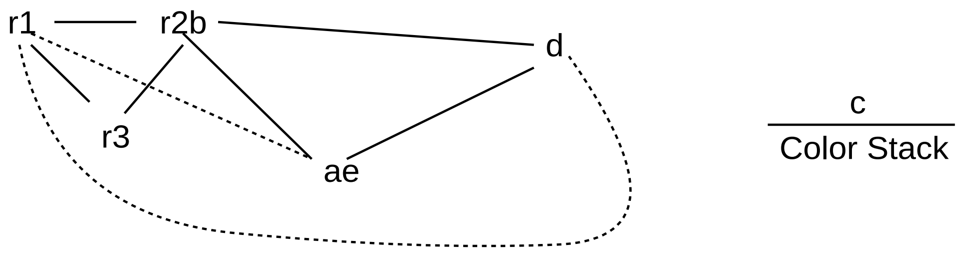
FR-373: **Complete Example**



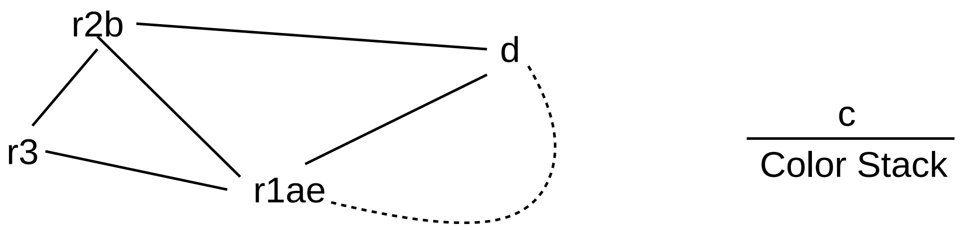
FR-374: **Complete Example**



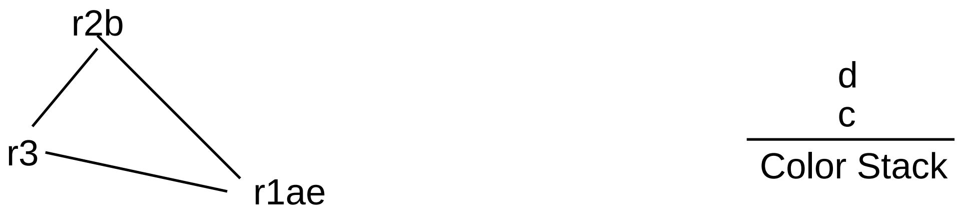
FR-375: **Complete Example**



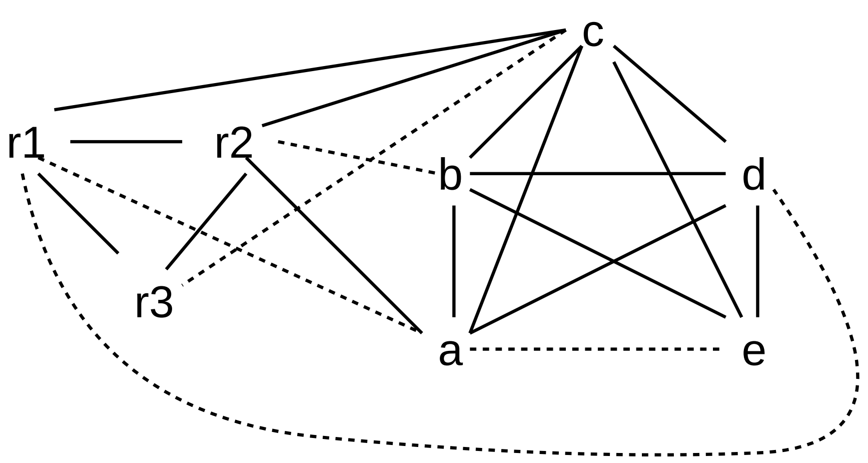
FR-376: **Complete Example**



FR-377: **Complete Example**

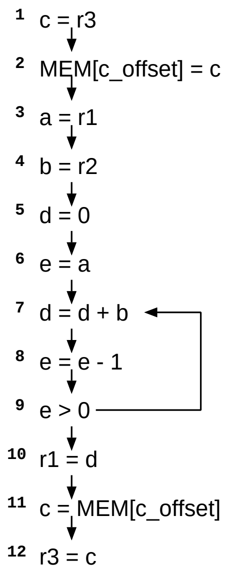


FR-378: **Complete Example**



FR-379: **Complete Example**

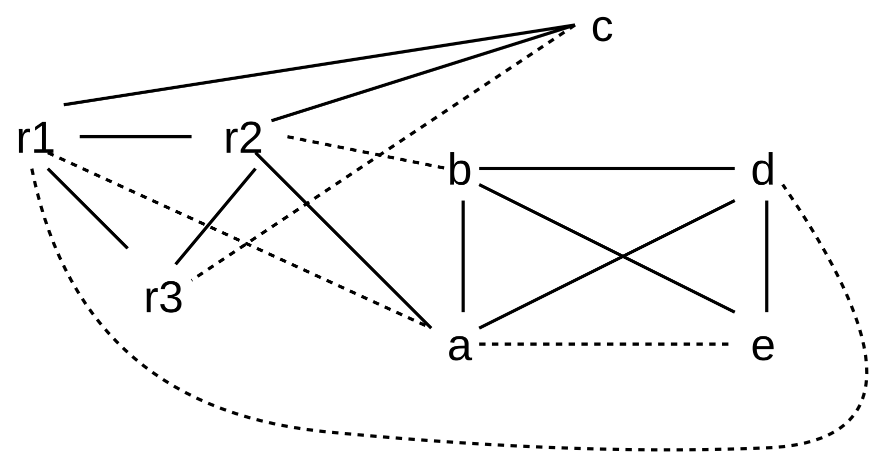
```
     c = r3
     MEM[c_offset] = c
     a = r1
     b = r2
     d = 0
     e = a
loop:
     d = d + b
     e = e - 1
    if e > 0 goto loop
    r1 = d
    c = MEM[C_offset]
    r3 = c
```



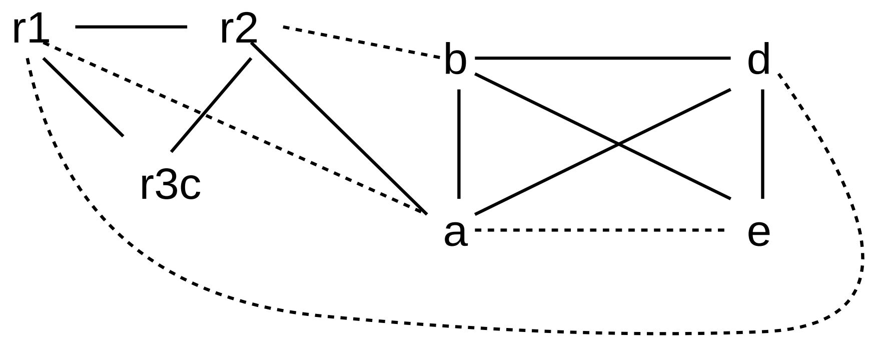
FR-380: **Complete Example**

| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | {r3} | {c} | {r1, r2, r3} | {c, r1, r2} |
| 2 | {c} | { } | {c, r1, r2,} | {r1, r2} |
| 3 | {r1} | {a} | {r1, r2} | {a, r2} |
| 4 | {r2} | {b} | {a,r2} | {a, b} |
| 5 | { } | {d} | {a, b} | {a, b, d} |
| 6 | {a} | {e} | {a, b, d} | {b, d, e} |
| 7 | {b, d} | {d} | {b, d, e} | {b, d, e} |
| 8 | {e} | {e} | {b, d, e} | {b, d, e} |
| 9 | {e} | { } | {b, d, e} | {b, d, e} |
| 10 | {d} | {r1} | {d} | {r1} |
| 11 | {} | {c} | {r1} | {c, r1} |
| 12 | {c} | {r3} | {c, r1} | {r1, r3} |

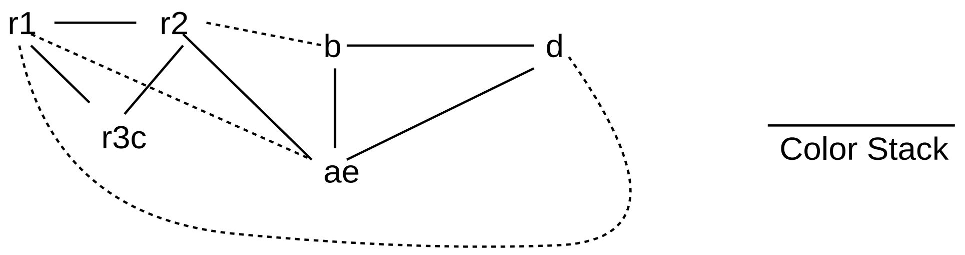
FR-381: **Complete Example**



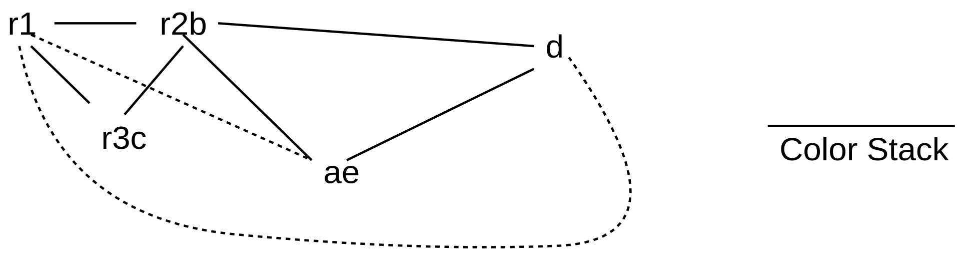
FR-382: **Complete Example**



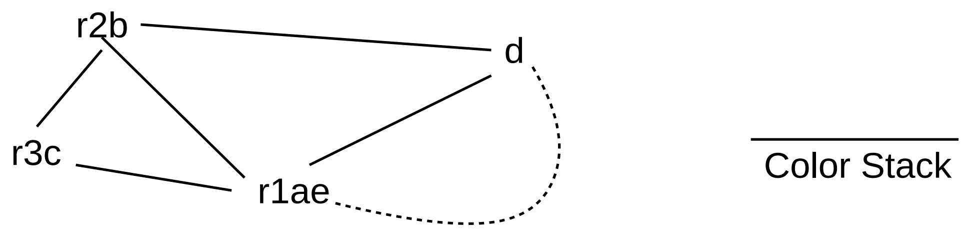
FR-383: **Complete Example**



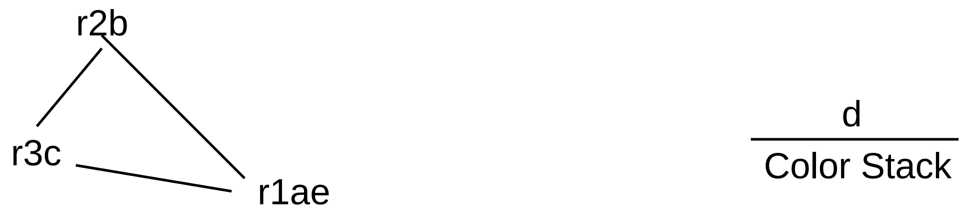
FR-384: **Complete Example**



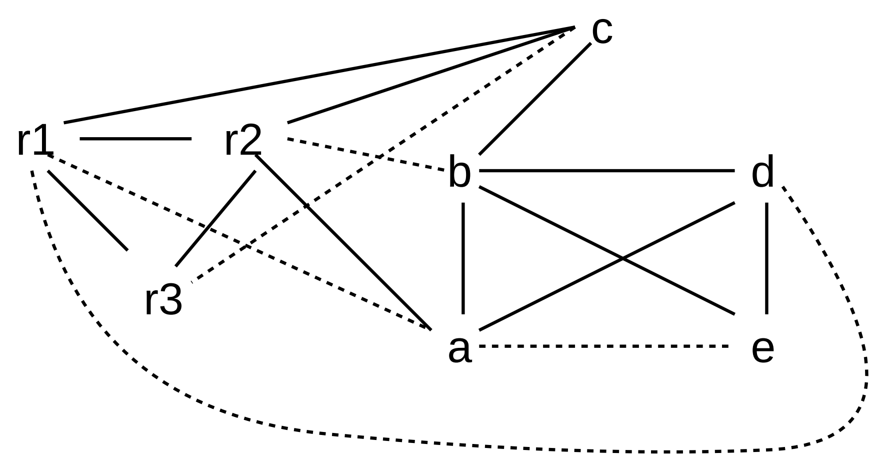
FR-385: **Complete Example**



FR-386: **Complete Example**



FR-387: **Complete Example**



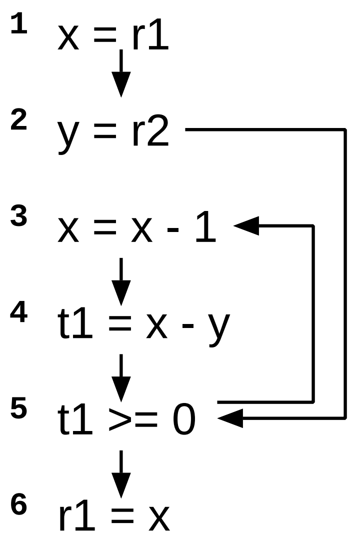
FR-388: **More examples**

```
int mod(int x, int y) {       mod:
   while (x >= y)                  x = r1
       x = x - y;                  y = r2
   return x;                       goto test
                              body:
}                                  x = x - 1
                              test:
                                   t1 = x - y
                                   if (t1 >= 0) goto body
                                   r1 = x
```

- r1, r2 are parameter registers
- r1 is (also) the result register

FR-389: **More examples**

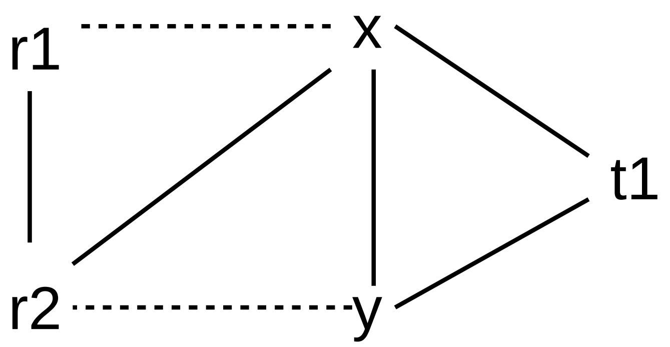
```
mod:
        x = r1
        y = r2
        goto test
body:
        x = x - 1
test
        t1 = x - y
        if (t1>=0) goto body
        r1 = x
```



FR-390: **More examples**

| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | {r1} | {x} | {r1, r2} | {x, r2 } |
| 2 | {r2} | {y} | {x, r2} | {x, y} |
| 3 | {x} | {x} | {x, y} | {x, y} |
| 4 | {x, y} | {t1} | {x, y} | {x, t1} |
| 5 | {t1} | { } | {x, t1} | {x} |
| 6 | {x} | {r1} | {x} | { } |

FR-391: **More examples**



FR-392: **More examples**

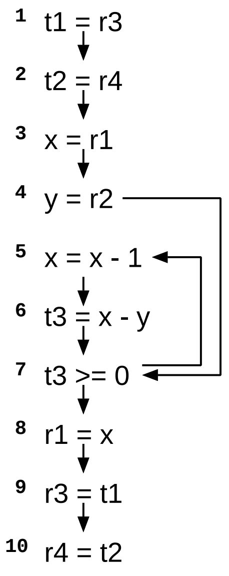
```
int mod(int x, int y) {      mod:
   while (x >= y)                t1 = r3
       x = x - y                 t2 = r3
   return x                      x = r1
                                 y = r2
                                 goto test
                             body:
}                                x = x - 1
                             test:
                                 t3 = x - y
                                 if (t3 >= 0) goto body
                                 r1 = x
                                 r3 = t1
                                 r4 = t2
```

- r1, r2 are parameter registers
- r1 is (also) the result register

- r3, r4 are callee-save

FR-393: **More examples**

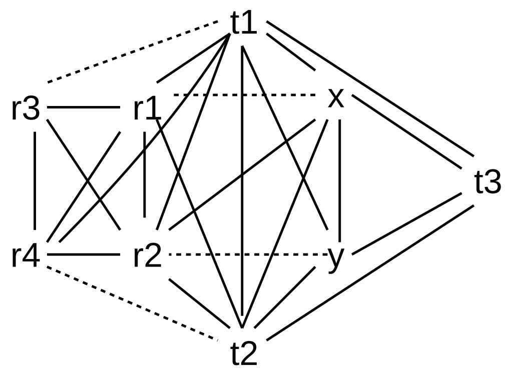
```
mod:
      t1 = r3
      t2 = r4
      x = r1
      y = r2
      goto test
body:
      x = x - 1
test
      t3 = x - y
      if (t3>=0) goto body
      r1 = x
      r3 = t1
      r4 = t2
```



FR-394: **More examples**

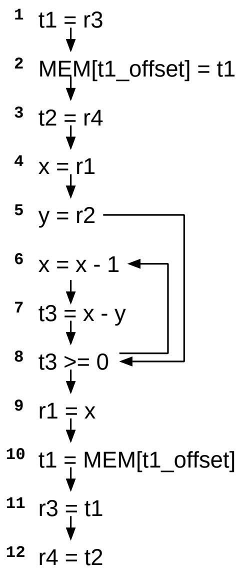
| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | {r3} | {t1} | {r1,r2,r3,r4} | {t1,r1,r2,r4} |
| 2 | {r4} | {t2} | {t1,r1,r2,r4} | {t1,t2,r1,r2} |
| 3 | {r1} | {x} | {t1,t2,r1,r2} | {t1,t2,x,r2} |
| 4 | {r2} | {y} | {t1,t2,x,r2} | {t1,t2,x,y} |
| 5 | {x} | {x} | {t1,t2,x,y} | {t1,t2,x,y} |
| 6 | {x,y} | {t3} | {t1,t2,x,y} | {t1,t2,t3,x,y } |
| 7 | {t1} | { } | {t1,t2,t3,x,y} | {t1,t2,x} |
| 8 | {x} | {r1} | {t1,t2,x} | {t1,t2} |
| 9 | {t1} | {r3} | {t1,t2} | {t2} |
| 10 | {t2} | {r4} | {t2} | { } |

FR-395: **More examples**



FR-396: **More examples**

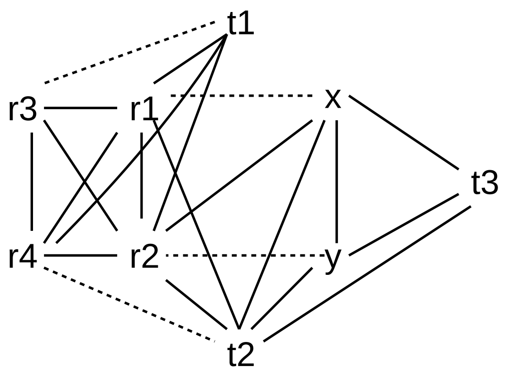
```
mod:
      t1 = r3
      MEM[t1_offset] = t1
      t2 = r4
      x = r1
      y = r2
      goto test
body:
      x = x - 1
test
      t3 = x - y
      if (t3>=0) goto body
      r1 = x
      t1 = MEM[t1_offset]
      r3 = t1
      r4 = t2
```



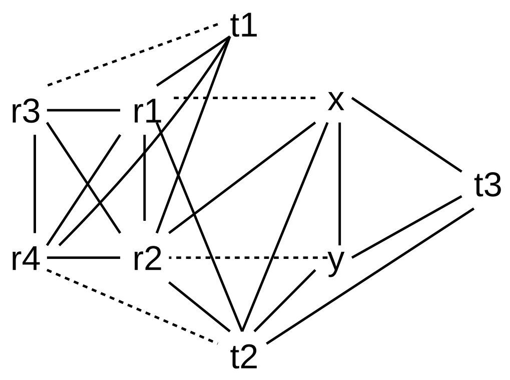
FR-397: **More examples**

| State | Use | Define | In | Out |
|---|---|---|---|---|
| 1 | {r3} | {t1} | {r1,r2,r3,r4} | {t1,r1,r2,r4} |
| 2 | {t1} | { } | {t1,r1,r2,r4} | {r1,r2,r4} |
| 3 | {r4} | {t2} | {r1,r2,r4} | {t2,r1,r2} |
| 4 | {r1} | {x} | {t2,r1,r2} | {t2,x,r2} |
| 5 | {r2} | {y} | {t2,x,r2} | {t2,x,y} |
| 6 | {x} | {x} | {t2,x,y} | {t2,x,y} |
| 7 | {x,y} | {t3} | {t2,x,y} | {t2,t3,x,y } |
| 8 | {t1} | { } | {t2,t3,x,y} | {t2,x} |
| 9 | {x} | {r1} | {t2,x} | {t2} |
| 10 | {t1} | { } | {t2} | {t1,t2} |
| 11 | {t1} | {r3} | {t1,t2} | {t2} |
| 12 | {t2} | {r4} | {t2} | { } |

FR-398: **More examples**



FR-399: **More examples**



FR-400: **More examples**

```
mod:
  t1 = r3
  MEM[t1_offset] = t1
  t2 = r4
  x = r1
  y = r2
  goto test
body:
   x = x - 1
test:
  t3 = x - y
  if (t3 >= 0) goto body
  r1 = x
  MEM[t1_offset] = t1
  r3 = t1
  r4 = t2
```

FR-401: **More examples**

```
mod:
  r3 = r3
  MEM[t1_offset] = r3
  r4 = r4
  r1 = r1
  r2 = r2
  goto test
body:
  r1 = r1 - 1
test:
  r3 = r1 - r2
  if (r3 >= 0) goto body
  r1 = r1
  MEM[t1_offset] = r3
  r3 = r3
  r4 = r4
```

FR-402: **More examples**

```
mod:
  MEM[t1_offset] = r3
  goto test
body:
  r1 = r1 - 1
test:
  r3 = r1 - r2
  if (r3 >= 0) goto body
  MEM[t1_offset] = r3
```

FR-403: **Calling functions**

- Any function call (jump to register)
  - Defines all callee-saved registers
  - Defines all paramter regiters
  - Defines result register
- All necesary saving of registers to the stack then happens automatically