Parse Tree - Evaluation

- Evaluation gives the value of the expression by traversing the tree
- Post-order traversal
  - recursive tree traversal algorithm
  - evaluate node after evaluating children
  - If you are looking at node N, evaluate the left subtree, then the right subtree, then the node.
Post-order Traversal

Postorder (Left, Right, Root) : 4 5 2 3 1
Semantic Analysis

• Check extra semantic information that is not handled in just checking if the token sequence follows the rules of the grammar.

• Type checking is one such thing - e.g. it is valid to say "varName1 = varName2" as per the rules of the grammar, but what if varName1 is an integer and varName2 is a string?

• Also, does varName1 even exist at this point in the program? Has it been declared yet?

• Build the symbol table that keeps track of what identifiers have been declared in the current scope, and what their type signatures are.
Code Generation

• Generate the code from the parse tree

• Eg.
• “10*(1+2)”

• CFG

• <expr> -> <expr> + <term> | <expr> - <term> | <term>
• <term> -> <term> * <fact> | <term> / <fact> | <fact>
• <fact> -> ( <expr> ) | int
• After, parsing the parse tree looks like this:

The simplified tree after removing non-terminals looks like this:
• Code generation for this will be:

```
addi $8, $0, 10
addi $9, $0, 1
addi $10, $0, 2
addi $9, $9, $10
mult $8, $9
; Result is in lo register.
```
Code generation

• Code generation becomes slightly more tricky when it comes to using variables, generating assembly for loops, function calls, classes, etc.

• In the end though it is just finding a general form for the high level programming concept, and applying it in the given context.

• Compilation finishes after the target language code is generated (usually assembly language). Next stage is assembler.
Assembler – already covered

- line by line translation
  - from **one** assembly instruction
  - to **one** machine code instruction
  - translate pseudo-instructions
- insert data for .**word** directive
  - and possibly other directives...
- ignore comments and blank lines
- compute address of each label
- Results in an “object” file
Linker

- Goal: combine multiple object files
  - modularity
  - avoid compiling whole program each time, only compile parts that changed
- Resolve external symbols - labels can refer to other object files. E.g. we could jump to a subroutine label that is NOT in our current object file.
- Produce the final executable file
Object File Format – Basics

- file header – meta information
- text segment – code
- data segment – static data
- defined external symbols
  - other objects files can refer to these labels
- undefined external symbols
  - Symbols not defined here, these labels must be found in other object files
- local symbols (for debugging, relocation)
Relocation

- An Assembler produces object code starting at address 0. How do combine multiple such object files?

- Relative addresses (e.g. branch) not a problem

- “Absolute” addresses, static data, must be fixed
  - Have the object file contains list of such code locations

- Adjust actual addresses in object code
Symbol Resolution

- replace symbol names with address
- labels/symbols must be unique
  - across all linked object files
  - manual name management
Imagine when we combine the object files produced by these two files that they are essentially stacked on top of each other. Then what was instruction 0 in File B now becomes instruction 100, and `mySubRoutine` is not located at instruction 42, but rather instruction 142. The linking process must then fix up the binary representation of the `jal` instruction so that it is jumping to the correct address (142*4).
Library

- collection of object files
- Contain functionality used by many programs (e.g. square root)
- ready for linking with other object files
Loading

- Before executing the object file, it needs to be loaded from the disk
- set up memory region(s) for new program
- load executable file from disk
  - perform late relocation and symbol resolution
  - see next slide
- create new process in operating system
- start process
Dynamic Linking

- dynamic linking
  - relocate and resolve symbols at load time instead of during linking

- dynamic library
  - do not add object code to executable file, combine object code at load time

- shared library
  - keep only one copy of object code in memory
  - Store it in a special memory area or relocatable object code
Dynamic Shared Library

- “dynamic link library” (DLL) on Windows
- modifications apply to all programs
  - no rebuilding necessary
DFA example

• A turnstile is locked initially. A customer puts a coin in to unlock it. The customer then pushes the arm to enter through it, after which the turnstile locks again. Pushing the handle when turnstile is locked does nothing. Putting in more coins when the turnstile is unlocked does nothing.

• DFA?