Comp 490 Notes: Compiler

**Video 1:**

**How Programming Languages Run**

* Interpreted Style
  + Takes input program and program input and outputs answer
* Compiled Style
  + Takes input program and translates it to a lower level language(machine code) to later be used in an interpreter.
* Compilers Phases
  + 1:Tokenizer/Lexer
    - Tokens: if, While,for examples. Break down words into units. Giving special meaning for the tokens.
  + 2:Parser:
    - Fed the tokens. Groups tokens together represent program instead into a tree like data structure.
    - Output = Abstract Syntax Tree (AST)
    - if(1 < 2){ return 7;}else{return 3;}

If

/ \ \

< return return

/ \ / \

1 2 7 3

* + 3:TypeChecker:
    - Reports if there is an error in meaning would reject programs that don’t make anysense
    - If a complex version will output an Annotated AST
  + 4:Code Generator:
    - Outputs Program Language
    - Can be broken into many parts depending out the goal and language
* Languages
  + //Grammar/ BNF/ Backus-Naur Form/ Context-free Grammar
  + Digit ::= ‘0’| ‘1’
  + Number::= digit | digit number
  + Expression::=number | expression ‘+’ expression
  + Example numbers
    - 0
    - 1
    - 01
    - 101
    - Rules above rules for any binary number.
  + Example Expressions
    - 1101
    - 101 + 110
    - (expression + expression) +(expression)
      * (1 + 111)+ (expression + expression) = (1 +111) + (01 + 10)
* Language design:
  + Integers & booleans
  + Declare and initialize variables
  + Perform typical arithmetic/logical operations
  + (Have to decide what we need )
  + Var is a variable
  + Num is a Number
  + Type::= ‘int’ | ‘bool’ —-> production rule is the full line
  + expression::= num|’true’|’false’|var|

‘(‘ op expression expression ‘)’

* + loop::= ‘(‘ ‘While’ expression statement ‘)’
  + statement::= vardec | loop| assign
  + op::= ‘+’ | ‘-’| ‘&&’| ‘||’ | ‘<’
  + program::= statement\*
  + “Int’ is an int production
  + Vardec ::= ‘(‘ ‘vardec’ type var expression ‘)’
    - (vardec int
    - (vardec bool
    - \*zero or more
    - Epsilon empty
* Object Langue(our language): We get to name this
* The Target Language is what you will compile to.
* Metalanguage(what we are writing the compiler in)
* Jacoco - shows what codes are touched in the tests.
* Junit-tests the code

VIDEO 2

Tokenization/Lexing

* Examples
  + ifToken
  + LeftParenToken
  + NumberToken(34)
  + LessThanToken
  + NumberToken(57)
* Comments get stripped away
* Define tokens after defining the grammar
  + Using previous grammar
    - IdentifierToken(String)
    - NumberTOken(int)
    - IntToken
    - BoolToken
    - …
* Goal can take an input program and convert into the tokens.
* Hashcode
* @override equal method
* Tostring
* Reserved words without context mean one thing.

Video3:

Parsing:

* Go from tokens to abstract syntax tree
* A parser takes the tokens and makes he syntax tree
* Literals have their own values
* Interface Type
  + Class IntType
  + Class BoolType
* interfaceStmt
  + Class vardecStmt
  + Class loopStmt
  + Class AssignStmt
* Interface Exp
  + Class NumberLiteralExp
  + Class BooleanLiteralExp
  + Class VariableExp
  + Class BinaryOperatorExp
* Interface op
  + Class PlusOp
  + Class MinusOp
  + Class LogicalAndOp
  + Class LogicalOROP
  + Class LessThanOP
* Class Program
* Parsing: the solved problem that isn’t
* Recursive descent parsing
* Parser generators(ANTLR)
* Parser combinators
* Why Paranthesis????
  + Helps with parsing
  + S-expressions: Lisp, Scheme, Racket, Clojure
    - (while(<7 4)
    - (while(<3 2)
  + Precedence isn’t needed for S-expressions
    - 1 + 3 \* 2 —-> (+ 1 (\* 3 2))
* Video4:
* TYPECHECKING
  + Types: used to describe data and the operations applicable to that data
  + Program with type errors: ill-typed
  + Program without any type errors:well-typed
  + Vardec puts a variable in scope with a type
    - Needs to remember the variable and the type
    - Need to ensure the expression is of the type
  + Num should be an int
  + True and false should be bools
  + Var is whatever the type of the variable is
  + While’s expression is a boolean
  + Assign
    - Var should be in scope
    - Var’s should match expression’s type
  + Exp1 +exp2 => int, exp1: int,exp2:int
  + …

Proteus Notes:

* Other HSMs generate code automatically from modeling tools that barely resembles the visualization. Makes it hard to analyze
* Other use DSLs(Domain-specific Languages)
  + Scala based—>only for simulations
    - Relies on garbage collection precludes it from the real-time embedded software
  + C-based DSL—--> simulations and real times
    - No safety guarantees, risk to mission-critical software
* Designed to look and behave like c/C++
* Built to focus on memory safety, and can be used in real-time systems.
* Previously built HSM developed from JPL help test to see if Proteus can implement them in a simplistic reasonable way. If not adjustments are made.
* Actor Model
  + Enables parallel Programming by splitting computation
  + Each actor executes code sequentially
  + Maintains its own executable code and maintains internal states upon which this code acts
  + Actors can’t manipulate other actor’s states
  + Input events that transition HSMs can be seen as messages for the actors
  + All actors are started at program start
  + All actors are terminated at program end
  + Actors are statically known and can’t be dynamically started or terminated at runtime.
  + Actor’s communicate with each other via messages.
  + Event is a reserved word
  + Actor’s have to fully process one event at a time. All others are stored in an internal event queue.
* Statemachine’s and states are declared
* Also the initial state needs to be indicated with ‘initial’
* Actors execute state machines sequentially. One at a time
* States can be nested in other states
* The most deeply-nested state currently being executed is the active state.
* If a child state’s parent can handle an event then while in the child state(that is nested int he parent state) can handle the event found in the parent(outer).
* State transitions are performed via ‘go’.
  + ‘go ‘ specifies the state to transition to.
* Doesn’t use Void
* Doesn’t use pointers
* Doesn’t use arrays
* Not an object-oriented
* ‘Entry’ ‘exit’ are reserved words
* Prints out trace lines to follow where the messages are going to and where and where they are from .
* Code
  + Enter/Exit
    - The actor is entering or leaving a state
  + Send{event} to {Actor}
    - Sent to actor’s event queue

Rotation 7 things

10th tuesday -1

12th thursday - 2

15th sunday - 3

17th tuesday - 4

19th thur - 5

22nd sun - 6

24th tues - 7