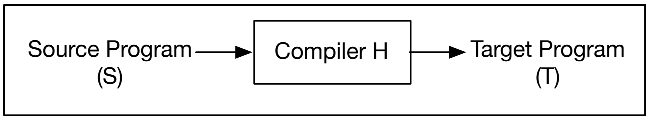
Thursday, October 29, 2015

Review

CS536

1. Compiler

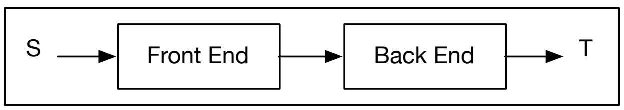


(1) Recognizer of language S

(2) Translator from S to T

(3) Program in language H

2.



(1) Front End: understand source code S

(2) IR: intermediate representation

(3) Back End: map IR to T

Finite State Machine

1. Scanner: translate sequence of chars *from source program* into *sequence of tokens*

INPUT OUTPUT

(1) Actions

- Group chars into lexemes(tokens)
- Identify and ignore whitecaps, comments, etc.

(2) Error checking

- *Bad* character
- *Unterminated* strings
- too large *literals*

(3) Each time call the scanner

- Find longest sequence of chars corresponding to a token —> return that token

2. Scanner generator

Regular Expression —> for each token & things to ignore(white-spaces, comments, etc.)

3. **Finite State Machine** (aka finite automata): [recognizer] represent regular languages

(1) Decide whether *string(sequence of chars)* is accepted/rejected

INPUT OUTPUT

(2) FSMs, formally (Q, Σ, δ, q, F)

- Q: finite set of states
- ∑: the alphabet (characters)
- δ: transition relation $\delta : Q \times \Sigma \longrightarrow Q$
- q: start state $q \in Q$
- F: final states $F \subseteq Q$

- FSM accepts string $x_1x_2x_3,\ldots,x_n \rightleftarrows \delta(\delta(\delta(q,x_1),x_2),x_3),\ldots,x_n)$
- The language of FSM M is the set of all words it accepts - L(M)

(3) FSM types

- **Deterministic**: no state has >1 outgoing edge with same label
- **Non-deterministic**: states may have multiple outgoing edges with same label

- Advantages of NFA: Much more compact (less states)

4. Recap

- The scanner reads stream of characters
- Using regular expressions —> finite state machine

Non-deterministic State Machine

1. NFA, formally (Q, Σ, δ, q, F)

- Q: finite set of states
- ∑: the alphabet (characters)
- δ: transition relation $\delta: Q \times \Sigma \longrightarrow 2^{|Q|}$
- q: start state $q \in Q$
- F: final states $F \subseteq Q$

2. Claim:

q: start state $!q \in Q$

2. Claim: L(NFA) = L(DFA) (NFA adds no power on DFA)

- Subset construction: each state of constructed DFA —> a set of NFA states

- in finitely many subsets of states at any time: !2|Q|

- Let ***succ(s, c)*** be the set of choices the NFA could make in state s with character c

- Build new DFA M' where Q' = $2^{|Q|}$

Add an edge from state S on character c to state S' - (the union of states that all states in S could possibly transition to on input c)

3. ε-transitions: Add no expressiveness to NFAs

- construct equivalent ε-free FSM

- εclose(s): set of all states reachable from s in zero or more ε-transitions

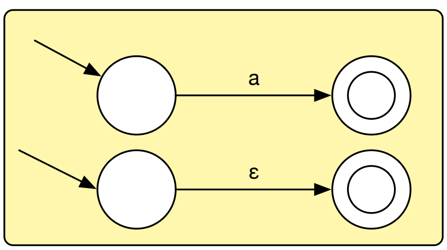
- 1) make s an accepting state of M if and only iff εclose(s) contains an accepting state

- 2) put $s, c \longrightarrow t$ in transition relation of M' iff there is a $q, c \longrightarrow t$ for some q in εclose(s)

Regular Expressions

1. Regular Expressions: pattern describing a language —> **tokenization**

- Operands: literal(single character)/epsilon —> simple DFA



- Operators: from low to high precedence —> methods of joining DFAs

precedence: | < . < *

- | (or = alternation)

- * (zero or more character)

- + (one or more character)

- . (any character)

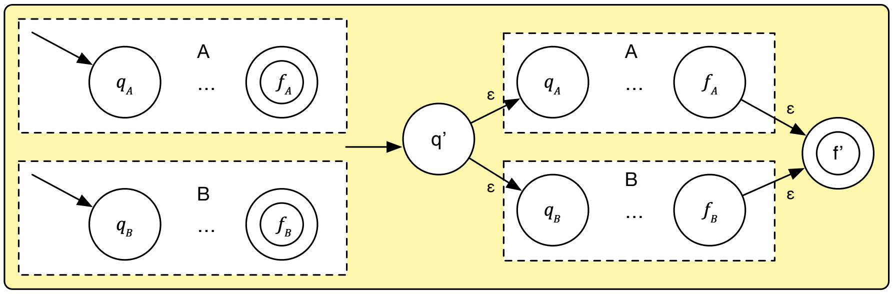
2. Regular expressions —> Non-deterministic Finite Automata

- q': new start state

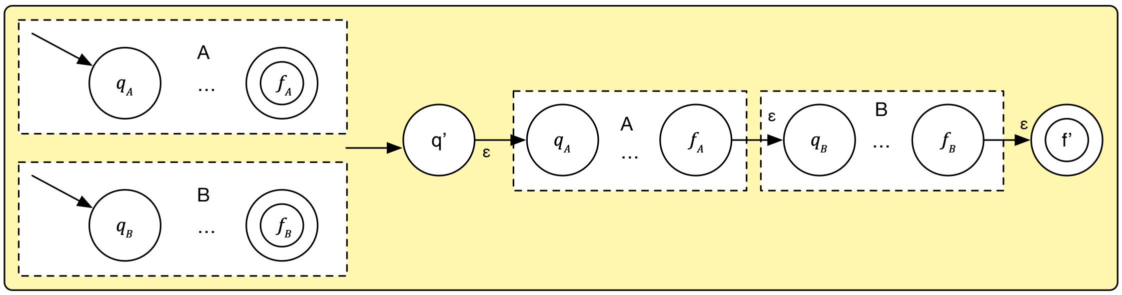
- f ': new final state

- original final states non-final

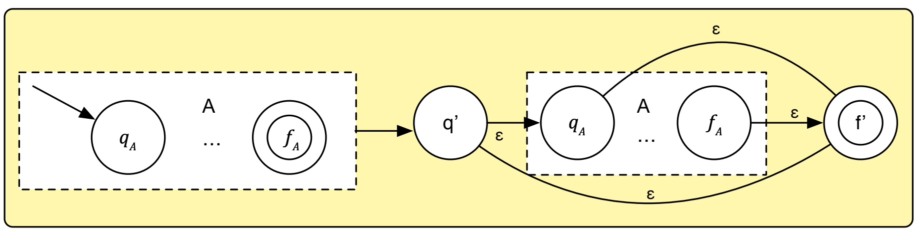
- Alternation A|B



- Catenation



- Iteration (A)*



2. Tree representation of a regular expression

- **Bottom-up conversion**

3. Regular expression —> DFAs

- RegEx —> NFA with ε —> NFA without ε —> DFA

4. <u>Table-driven DFAs</u>

- δ can be expressed as a table —> efficient array representation

5. FSMs for tokenization

- FSMs only check for language membership of a string
  - Scanner needs to recognize a stream of many different ***<u>tokens</u>*** using the <u>longest match</u>
  - Scanner needs to know ***<u>what</u>*** was matched
- FSM can <u>peek</u> at characters past the end of a valid token —> convenient to add an EOF symbol

6. **Lexical analyzer generators** (aka scanner generators)

- Transformation from regular expression to scanner is formally defined
- Synthesize a lecture automatically: Lex, Flex, JLex

- JLex: declarative specification

set of regular expressions + associated actions —> Java source code for a scanner

INPUT OUTPUT

- JLex format: 3 sections separated by %%

(1) User code section

(2) Directives: macro definition, state declaration

(3) Regular expressions + actions

- <regex> {code} —> <regex> can include {macros} from directives section
- chars & "chars" —> represent themselves especially special characters
- Regular expression operators: | * + ? () .
- Character class operator: - (range) ^ (not) \ (escape)

Context-free Grammars

1. Limitations of regular expressions

- matching cannot be handled (Eg. balanced parentheses)
- Cannot enforce the order of operations with a stream of tokens

2. Context free Grammar (CFG)

- A set of (recursive) rewriting rules to generate patterns of strings
- Envision a "parse tree" that keeps structure

(1) rewrite rule !S− >(S): rewrite S to be an S surrounded by a single set of parentheses

(2) CFG recognize the language of **trees** where all leaves are terminals

(3) 4-tuple CFG —> !G = (N , Σ,P, S)

- N: the set of **nonterminal** symbols [impose a hierarchical structure]- placeholder/interior nodes in the parse tree
- Σ: the set of **terminal** symbols - tokens from scanner
- P: the set of **productions** - rules of deriving strings
- S: the start nonterminal in N - the non-terminal that appears on the LHS of 1st production

- Production syntax: LHS → RHS ≡ Nonterm → expression|ε —> define the ***syntax*** of a language

- LHS: Single nonterminal symbols
- RHS: Expression - Sequence of terminals and nonterminals
- Notation: "BNF" or "enhanced BNF" (Backus-Naur Form)

3. Derivations

- Mechanism
- Start by setting "Current Sequence" to the start symbol
- Repeat: Find a nonterminal X in Current Sequence

  Find a production of the form !X $\rightarrow \alpha$

  Apply the production: create a new Current Sequence in which α replaces X
- Stop when there are no more nonterminals
- symbol $\Rightarrow$ for derives

$\overset{+}{\Rightarrow}$ derives in one or more steps; $\overset{*}{\Rightarrow}$ derives in zero or more steps

4. Formal CFG Language Definition L(G) = {w | S $\Rightarrow$ w}

- S: the start nonterminal of G
- w: a sequence of terminals or ε

5. List Grammar: **repeat** a structure arbitrarily often

- Derivation order (skew direction)
- **leftmost(rightmost) derivation**: always expand the leftmost(rightmost) nonterminal

6. Ambiguity - derive the same string in multiple ways even with a fixed derivation order

- G is **ambiguous** if
- >1 leftmost derivation of w
- >1 rightmost derivation of w
- >1 parse tree for w
- Resolving grammar ambiguity

(1) **Precedence**: nonterminals are the same for both operators

To fix precedence:

- one nonterminal per precedence level
- Parse lowest precedence level first

(2) Associativity

Recognize left-associative(right-associative) operators with left-associative(right-associative) productions

Recursion in Grammars

- A grammar is ***recursive*** in (nonterminal) X if $X \Rightarrow^{+} \alpha X \gamma$ for non-empty strings of symbols α and γ
- A grammar is ***left-recursive*** in X if $X \Rightarrow^{+} X\gamma$ for non-empty string of symbols γ
- A grammar is ***right-recursive*** in X if $X \Rightarrow^{+} \alpha X$ for non-empty string of symbols α

5. Makefiles

- record a series of commands in a script-like DSL(Domain Specific Language)
- Specify dependency rules & generates the results
- thread common configuration values through makefile
- <target>: <dependency list>

  <command to satisfy target>

Syntax Directed Translation

1. Syntax Directed Translation

- Augment CFG rules with translation rules (at least one per production)
- translation of LHS nonterminal
  - Constants + RHS nonterminal translation + RHS terminal value
- Assign rules bottom up

2. **Abstract Syntax Tree**

- A condensed form of the parse tree: syntactic details omitted
- Operators at internal nodes (not leaves)

(1) AST implementation —> ASTs in code

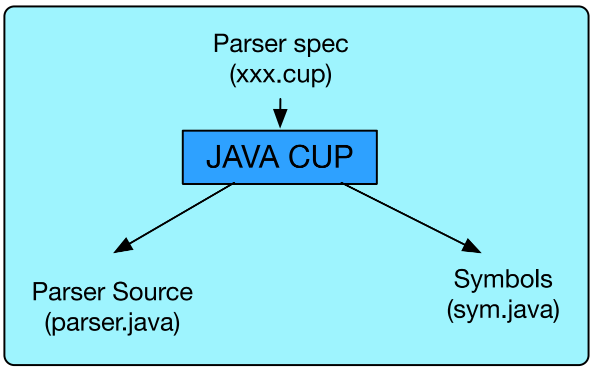
- Define classes for each type of nonterminal
- Create a new nonterminal in each rule

	Language abstraction	Output	Tool	Implementation
Scanner	RegEx	Token Stream	JLex	DFA walking via table
Parser	CFG	AST(Parse Tree)	Java Cup	

Java CUP

1. Parser Generator: guarantee that the program is structurally correct

- Input: an SDT (Syntax Directed Tree) spec
- Output: an AST (Abstract Syntax Tree)
- Tools: Java CUP



(1) Java CUP Input Spec

- Terminal & Nonterminal declarations
- Optional precedence and associativity declarations
- Grammar with rules and actions

(2) Parser.java

- Constructor: takes arguments of type ***Yylex***
- Contains parse method: return Symbol whose value contains translation of root nonterminal
- Uses output of JLex (Depends on scanner and token values)
- Uses defines of AST classes (ast.java)

Parsing - CYK in CNFs

1. CYK

- Bottom-up approach - “Data Driven”
- Time complexity: $O(n^3)$
- No problems with ambiguous grammars: gives a solution for all possible parse tree simultaneously
- Only takes grammar in **CNF** (Chomsky Normal Form)

- All rules must be one of two forms: (1) X -> t (2) X -> AB

- Only the start S is allowed to derive epsilon —> forbid the RHS of any rule

2. Implement CYK in CNF

- Nonterminals come in pairs —> subtree as a sub-span of the input
  - X—> t: Production form the leaves of the parse tree
  - X —> A B: Form binary nodes
- Eliminating Useless Nonterminals
  - If a nonterminal cannot derive a terminal symbol —> useless
  - If a nonterminal cannot derived from the start symbol —> useless

3. CNF in 4 steps

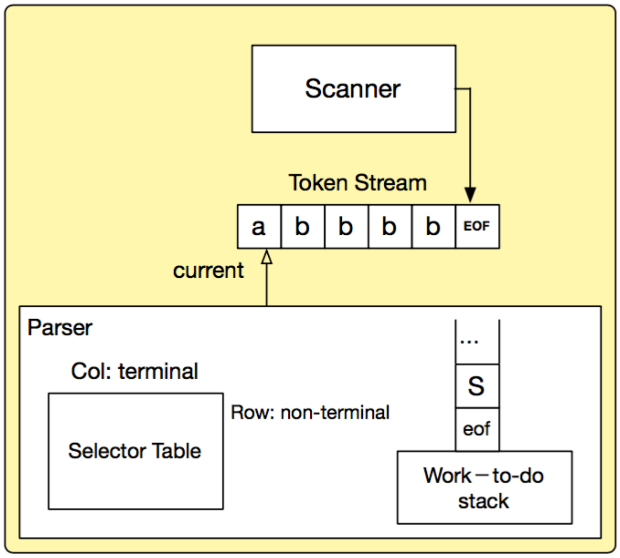
- **Eliminate epsilon rules**
  - Make copies of all rules with A on the RHS
  - Delete all combinations of A in those copies
- **Eliminate unit productions**: productions of the form A —> B
  - Place B anywhere A could have appeared
  - Remove the unit production
- **Fix productions with terminals on RHS**
  - For each terminal t add the rule X —> t
  - Replace t with X in the original rule
- **Fix productions with >2 non-terminal on RHS**
  - Replace all but the first nonterminal with a new nonterminal
  - Add a rule from the new nonterminal to the replaced nonterminal sequence
  - Repeat

Parsing - LL(1)

1. **Top-down** Parsers

- Start at the Start symbol
- **Predict** what productions to use

Thursday, October 29, 2015



2. LL(1) parsing

- O(n) parser
- Parseable by Predictive (top-down) parsers: recursive descent
- If **selector table** has <u>one production per cell</u> —> LL(1) grammar

(1) **LL(1) grammar transformation**: <u>necessary but not sufficient</u> for LL(1) Parsing

i. Free of **<u>left recursion</u>**: $X \Rightarrow X\alpha$

- **immediately left recursive**: $A \Rightarrow A\alpha \mid \beta$
- no <u>nonterminal loops</u> for a production: need to look past list to know when to cap it —> stack overflow

- General rules remove immediate left-recursion

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n \mid A\beta_1 \mid A\beta_2 \mid \dots \mid A\beta_n$ —> $A \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_2 A'$

$$A' \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_\mu A' \mid \varepsilon$$

ii. **<u>Left factored</u>**

- no rules with <u>common prefix</u>: need to look past the prefix to pick rule

- General rules to remove left factor

$A' \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_m \mid y_1 \mid \dots \mid y_n$ —> $A \rightarrow \alpha A' \mid y_1 \mid \dots \mid y_n$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$

(2) Build the **selector table**

- FIRST set - FIRST(α)
- Set of terminals that can begin at a subtree rooted at the arbitrary symbol α(derivable from α)

$\text{FIRST}(\alpha) = \{t \mid (t \in \Sigma \wedge \alpha \Rightarrow^* t\beta) \vee (t = \varepsilon \wedge \alpha \Rightarrow^* \varepsilon)\}$

- **<u>FIRST sets construction</u>**: single symbol

Begin: <u>single</u>, arbitrary symbol X

- If X is a terminal —> FIRST(X) = {X}
- If X is ε —> FIRST(ε) = {ε}
- If α is a nonterminal, FIRST(α) for each $\alpha = Y_1 Y_2 \ldots Y_k$
  - Add FIRST(Y1) - {ε}
  - If ε is in FIRST(t c$Y_{1 \text{ to } i\text{-}1}$): add FIRST($Y_i$) - {ε}
  - If ε is in all RHS symbols, add ε

- FOLLOW set - (A)

- Set of terminals that can appear immediately to the right of nonterminal A

$\text{FOLLOW}(A) = \{t \mid (t \in \Sigma \wedge S \Rightarrow^+ \alpha A t \beta) \vee (t = \text{eof} \wedge S \Rightarrow^+ \alpha A)\}$

- **<u>FOLLOW sets construction</u>**

FOLLOW(A) for X —> αAβ

- If A is start nonterminal, add eof
- Add FIRST(β) - {ε}
- Add FOLLOW(X) if ε in FIRST(β) or β is empty

- Selector table construction Table[X][t]

For each X —> α

- for terminal t in FIRST(α), put α in Table[X][t]
- if ε is in FIRST(α), for each terminal t in FOLLOW(X), put α in Table[X][t]

3. **Semantic stack (bottom-up SDT)**: hold <u>nonterminals' translation</u>

- LL (1): implicitly tracked via the semantic stack
- On <u>semantic stack</u>: SDT rules converted to SDT actions
  - Pop translations of RHS nonterminals off
  - Push computed translation of LHS nonterminals on
- Action number: define ***when*** to fire SDT actions
  - Number actions and add action number symbols <u>at the end of productions</u>
  - Placing: <u>after</u> their corresponding nonterminal & <u>before</u> corresponding terminal

Bottom-up Parsing - SLR(1)

1. LL(1) parsing: for simple parsing jobs

2. Top-down parser

- Parser operation
  - Scan the next input token
  - Push a bunch of RHS symbols
  - Pop a single symbol

3. Bottom-up parser

- Know exactly where we are & make predictions about next
- Parser operation
  - *Shift* an input token into a stack item
  - Reduce a bunch of a stack items into a new parent item

4. LR parser

- Left-to-right scan of the input file
- **Reverse rightmost derivations**: $S \Longrightarrow \alpha_1 \Longrightarrow \alpha_2 \Longrightarrow \ldots \Longrightarrow \omega$ (terminal string)
  - $\alpha A\gamma \Longrightarrow \alpha\beta\gamma$: a step in the derivation, so $A \rightarrow \beta$ is a production in the grammar
  - **LR(k)**: for every derivation step, $A \rightarrow B$ can be inferred using only a scan of $\alpha\beta$ and at most k symbols of γ
- Advantages
  - Can recognize almost any programming language
  - Time and space complexity: O(n)
  - More Powerful: LL(1) < LR(1)
- Disadvantages
  - More complex parser generation
  - Larger parse table

5. Parser state

- Top-down parser state
  - Maintains a **symbol stack** (represent what we expect in the rest of our descent to leaves) and **current token**
  - Works down and to the left through the tree
- Bottom-up parser state

- Maintains a **symbol stack** (represent summary of input we've seen) and **current token**
- Works upward and to the right through the tree
- Need an **auxiliary state machine** to help disambiguate rules

(1) LR(1): recognize any DCFG & can experience blowup in parse table size

(2) LALR(1)

(3) SLR(1): both proposed at the same time to limit parse table size

6. **Stack Item**: representative of symbols

- Indicate a production and a position within the production: $\mathbf{X \longrightarrow \alpha.B\beta}$
- May not know exactly which item you are parsing

(1) Build **LR Parser FSM**: track the set of states that you could have been in

(2) Automation as table

- *Shift*: taking a terminal edge
- *Reduce*: taking a nonterminal edge
  - When to reduce: Only see terminals in the input

7. Problem of LR parser: tracking sets of states can cause the size of FSM to blow up

- Small modification to LR's item and table form

(1) Two sets

- Closure(I): the set of items that could be mistaken for I

while there exists an item in Closure(I) of the form $X \longrightarrow \alpha.B\beta$, add $B \longrightarrow .\gamma$ if it'w not in Closure(I)

- Goto(I, X) = Closure($\{A \longrightarrow \alpha X.B \mid A \longrightarrow \alpha.X\beta$ is in I$\}$): currently in state I, the place after parsing X

(2) Parse table construction

- Add new Start S' and $S' \longrightarrow S$
- Build State I_0 for Closure($\{S' \longrightarrow .S\}$)
- Saturate FSM: for each symbol X s.t. there is an item in state j containing .X

(3) Build the Action Table: Reduction and Shift

- Efficient when at most one R (reduce) per cell

(4) Build the GoTo Table

Semantics

1. Semantic Error

- Fundamental undecidability problems: halting or crashes
- Practical feasibility: thread interleaving or inter-procedure data flow

2. Semantics: meaning of a program

- Parser: guarantee structural correctness

(1) Name analysis (aka. name resolution)

- Associate ids with their uses
- For each scope
  - Process declarations: Add them to symbol table
  - Process statements: update IDs to point to their entry

(2) Type analysis

Process statements: Use symbol table information to determine type of each expression

3. Symbol table: binds names to information we need

- Information needed in an entry

Kind (struct, variable, function, class)	Type	Nesting level	Runtime location (stored in memory)

- Given a declaration of a name, whether multiply declared in the current scope.
- Given a use of a name, it corresponds to which declaration or un-declared.
- Operations
  - Insert entry
  - Look-up name
  - Add scope (Add new table) ⟷ Remove scope (Remove/forget a table)

4. **Scope**: Block of code in which a name is visible/valid (lifetime of a name)

- Static vs. dynamic
  - **Static scope**: correspondence between use and declaration of a variable is known at compile time
  - **Dynamic scope**: correspondence is determined at runtime

⟶ use corresponds to declaration in **most-recently-called still active function**

- **Variable shadowing**: allow names to be reused in nesting relations, even with different types

- **Overloading**: same name for different methods either different type or different formals
- **Forward references**: use of a name before if is filled out in the symbol table
  - Requires two passes over the program (1st: fill symbol table; 2nd: use it)

5. name analysis implemented with an AST

- Walk the AST
  - Augment AST nodes with a link to the relevant name in the symbol table
  - Build new entires into the symbol table when a declaration is encountered

6. Type Checking

- Type - short for "data type"
  - Classification identifying kind of data
  - A set of possible values which a variable can possess
  - Operations that can be done on member values
  - A representation (perhaps in memory)
- **Components** of a type system
  - Primitive types + means of building aggregate types
  - Means of determine if types are compatible
  - Rules for inferring type of an expression
- **Type coercion**: implicit cast from one data type to another
  - Narrow form: type promotion - When the destination type can represent the source type

7. Typing

(1) ***When*** to check type

- **Static typing**: type checks are made before execution of the program
  - compile-time optimization
  - compile-time error checking
- **Dynamic typing**: type checks are made during execution (runtime) - add flexibility
- Combination of two:
  - down-casting (dynamic check)
  - cross-casting (static check)
- **Duck typing**: type is defined by methods and properties
  - Duck punching: runtime modifications to allow duck typing

(2) ***What*** to check

- Strong vs. weak typing
  - Degree to which type checks are performed
  - Degree to which type errors are allowed to happen at runtime
  - Continuum without precise definitions

Some not-universal definitions:

- Statically typed always stronger - fewer errors
- More implicit casts allowed always weaker

(3) Type safety

- All successful operations must be allowed by the type system
- Java - explicitly designed to be type safe
  - A variable is guaranteed to be of that of type when it is declared

---

Runtime environment

1. **Runtime environment**: underlying software and hardware configuration assumed by the program

- May include an OS & a virtual machine
- Role of **OS** (program piggybacks on OS)

- Provides functions to access hardware
- Provides illusion of uniqueness
- Enforces some boundaries on what is allowed

→ Meditation is slow

- **Compiler** - best use the runtime environment

- Limited number of very fast registers for computation
- Comparatively large region of memory to hold data
- Basic instructions from which to build more complex behaviors

2. General memory layout → program memory: a single array

- Addressable via memory cell (**function frame**): represented by **HEX** values
- **Goals** to divide up memory: flexibility, efficiency, and speed

3. Static allocation

- One **frame (slot)** for each subroutine (parameters and local variables)

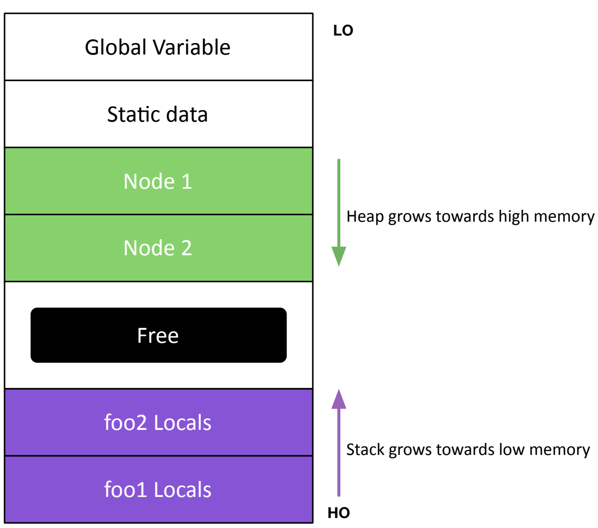
- **Advantages**: Fast access to all names & No computation overhead of stack/heap manipulation
- **Disadvantages**: No recursion & No dynamic allocation

4. **Stack allocation**: **Dynamic locals** : local variables of unknown size at compile time

- Dynamically allocate frame dynamic locals (currently active methods)
- Fix a pointer in memory: grows from the pointer during its execution
    - Store the previous frame's boundaries in the current frame
- Allocate frame per **activation record** (AR)
    - Push a new frame on function entry
    - Pop the frame on function exit
    - To keep size down → Put static data in the global area
- Two registers track the stack
    - **$fp** - frame pointer: tracks the base of the stack
    - $sp - stack pointer: tracks the top of the stack
- Stored in the frame:
    - Local variable value
    - Space for caller

(i) *Data context*: the boundaries of the caller frame

(ii) *Control context*: the line of code where we made the call



5. **Heap allocation**: non-local dynamic memory

- Not all data allocated in a function call to disappear on return

- Such objects are dynamically allocated ⟷ *Free* when it's unused: programmer specified or tracked automatically

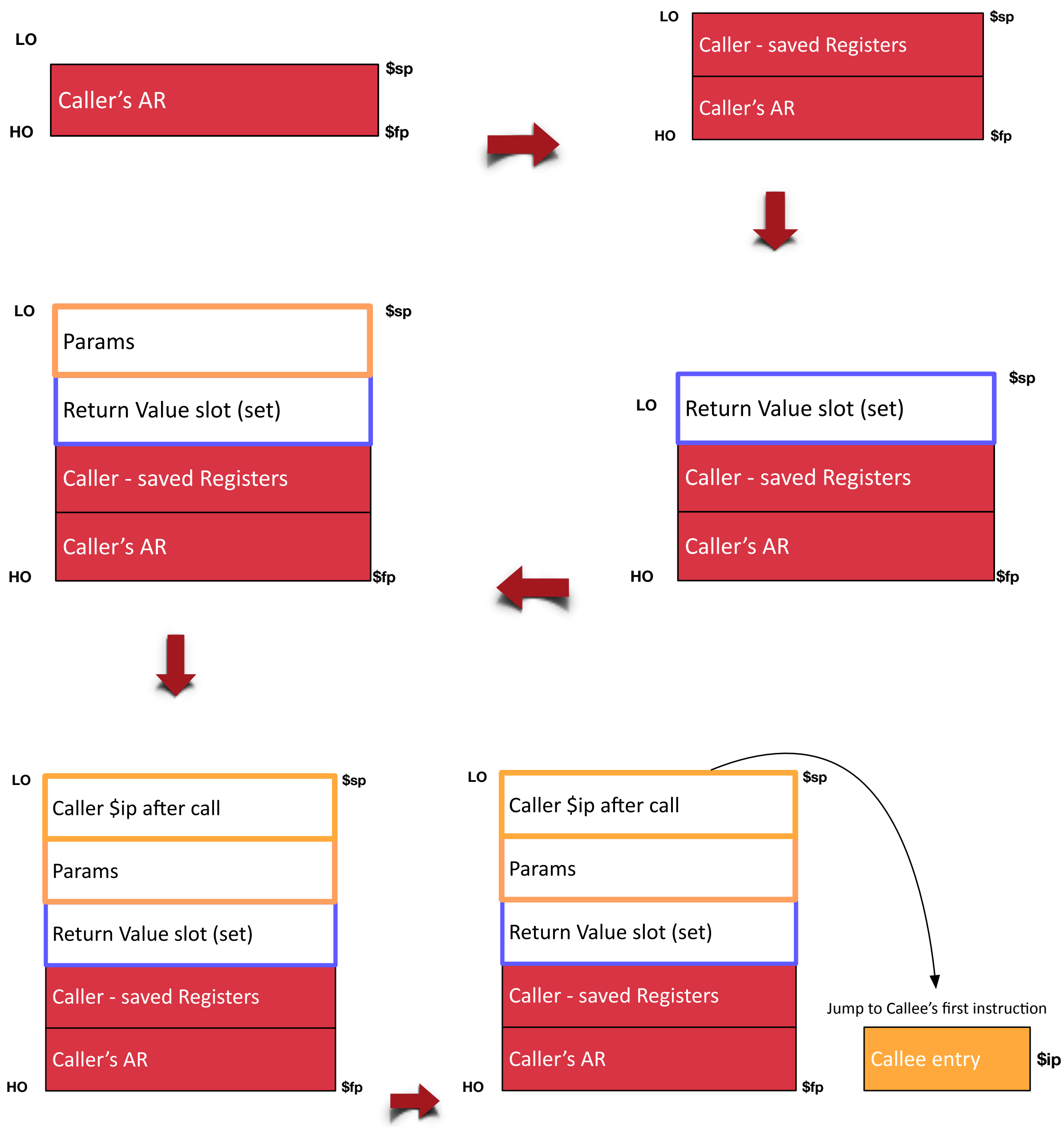
6. Function calls

- **$ip - instruction pointer**: tracks the line of code executing
- ***Caller***: function doing the invocation ⟷ ***Callee***: function being invoked

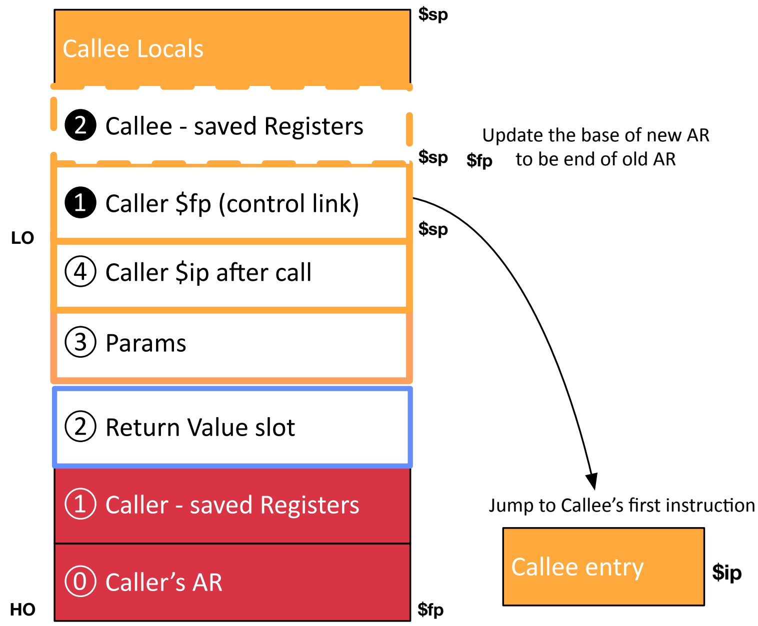
- Per call relationship

(1) Function entry

- Caller responsibility



- Callee responsibility



(2) Function exit

7.

8.