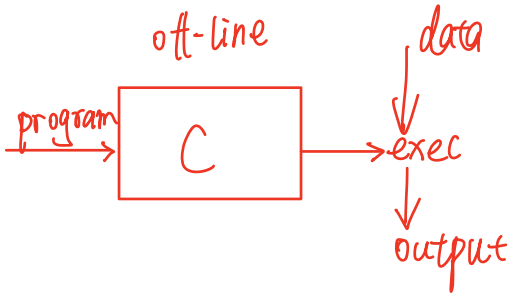
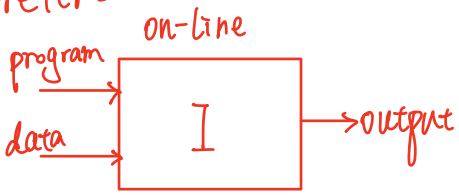
0101 Introduction:

① compilers:



② interpreter:



0102 Structure of compilers:

① lexical analysis:

recognize words — divide program text into "tokens"

② parsing

grouping tokens together according to some structure

③ semantic analysis

programming languages have strict rules to avoid ambiguities

④ optimization

automatically modify programs so that

- run faster
- use less memory

⑤ code generation

0103 the economy of programming languages:

0301 lexical analysis:

token classes correspond to sets of strings

① Identifier

strings of letters or digits, starting with a letter

② Integer:

a non-empty string of digits

③ keyword:

else, if, begin ...

④ whitespace:

a non-empty sequence of blanks, newlines, and tabs.

contigious white space will be treated a one token

in other words, token is <class/role, string>

0302 Lexical Analysis Examples

FORTRAN rule: whitespace is insignificant

① Look ahead:

e.g. FORTRAN
```
DO 5 I = 1,25
DO 5 I = 1.25
```

totally different meanings

so needs look ahead to determine meaning for token

sometimes needs unbounded-lookahead

② PL/I keywords are not reserved

0303 Regular Languages

① Regular Expression

results are sets:

a. single character: only contain 1 character

b. epsilon: only contain empty string

c. union: just merge 2 sets into 1 set

d. concatenation

e. iteration

could consider languages as a set of strings

② regular expressions over Σ (alphabet): smallest set of expressions including

0304 Formal Languages

① Let Σ be a set of characters (an alphabet).
a language over Σ is a set of strings of characters drawn from Σ

② meaning function: mapping syntax to semantics

L : Exp → a set of strings

many to one

0305 Lexical Specifications:

① keyword: 'if' + 'else' + 'then' + ⋯

② Integer:

digit = '0'+'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'

→ digit$^+$ = digit · digit*

③ identifier:

letter = 'a'+'b'+'c'+⋯ = [a-zA-Z]

→ letter · (letter+digit)*

④ whitespace: blanks, newlines, tabs

(' '+'\n'+'\t')$^+$

(x)+ε = (x)?

04이 Lexical Specification

① partition a string into a consecutive tokens

1. write a rexp for the lexemes of each token class
   e.g. number = digit$^+$

2. construct R, matching all lexemes for all tokens
   e.g. R = keyword + identifier + number + ···
   $= R_1 + R_2 + \cdots$

3. let input be $x_1 \cdots x_n$
   for $1 \le i \le n$ check
   $x_1 \cdots x_i \in L(R)$

4. if success, then we know that
   $x_1 \cdots x_i \in L(R)$ for some j

5. remove $x_1 \cdots x_i$ from input and go to (3)

② how much input is used?
   if 2 prefix of 1 string matches,
   choose the longgest matched token

③ Which token will be used?

1 prefix of string matches 2 different tokens, choose the one listed first (priority)

④ what if no rule matches?

no allowed $x_1 \cdots x_i \notin L(R)$ to happen

define a error class token with least priority matches tokens that does not match any previous tokens

0402 Finite Automata

① components:

a. An input alphet Σ

b. a finite set of states S

c. a start state n

d. a set of accepting states $F \subseteq S$

e. a set of transitions state $\xrightarrow{\text{input}}$ state

② Language of a FA: set of accepted strings

0403 Regular Expression into NFAs

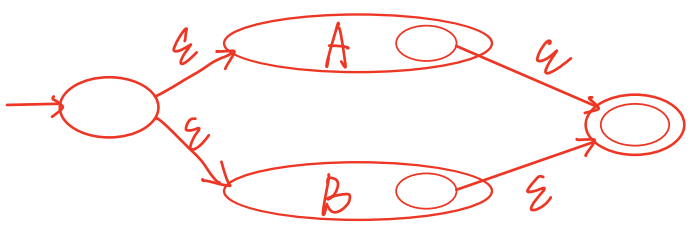
① for each kind of rexp, define an equivalent NFA

a. ε

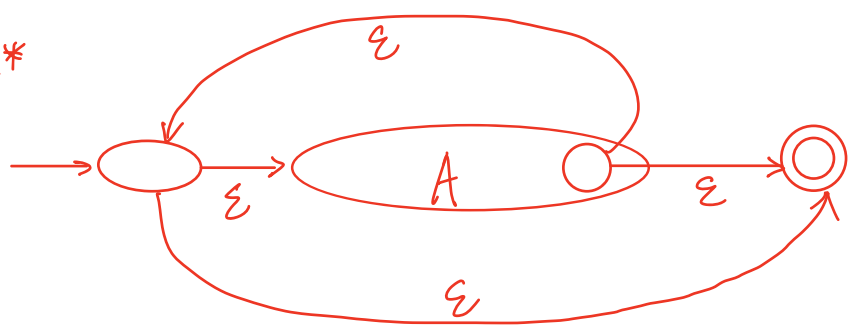
b. a

c. AB

d. A+B



e. A*



0404 NFA → DFA

① ε-closure of a state (ε-clos)
all states can be reached from a state

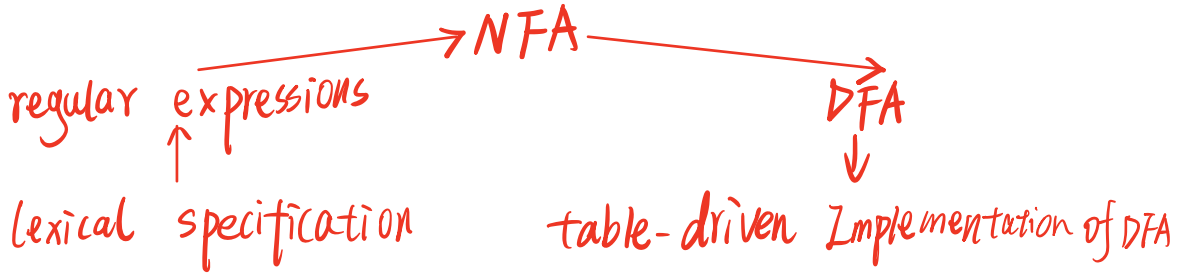
② $a(x) = \{y \mid x \in X_n, x \xrightarrow{a} y\}$
that is, if X is within given states X, return all states x can reach with input a.

③ DFA
a. states: Subsets of S except empty set
b. start state: ε-clos(s) (start state of NFA)
c. final start: $\{x \mid X \cap F \neq \emptyset\}$
↳ (final states of NFA)

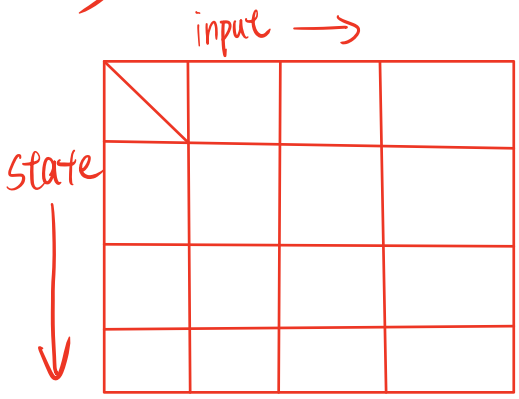
d. $X \xRightarrow{a} Y$ if $Y = \varepsilon\text{-clos}(a(x))$

0405 Implementing Finite Automata

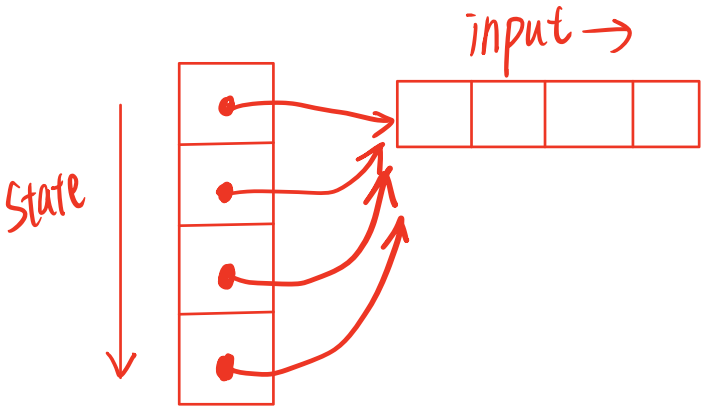
① steps of lexical analysis:



② a DFA can be implemented by a 2D table T
   a. one dimension is states
   b. other dimension is input symbol
   c. for every transition $S_i \xrightarrow{a} S_k$ define $T[i, a] = k$



memory efficient way:



③ table-driven implementation for NFA, column for ε is needed

④ trade-off: DFA vs NFA

possible $2^N - 1$ states

DFAs	faster,	less compact
NFAs	slower,	concise

05.01 Introduction to Parsing

① for a parser:

input: sequence of tokens from lexer

output: parse tree of the program

05.02 Context-Free Grammars

① components:

a. a set of terminals (T)

b. a set of non-terminals (N)

c. a start symbol S ($S \in N$)

d. a set of productions

$X \to Y_1 \cdots Y_N$

$X \in N$

$Y_i \in N \cup T \cup \{\varepsilon\}$

② procedures:

a. begin with a string with only the start symbol S

b. replace any non-terminal X in the string by the right-hand side of some production $X \to Y_1 \cdots Y_n$

c. repeat (b) until there are non-terminals

③ CFG:

Let G be a context-free grammar with start symbol S. Then the language L(G) of G is:

$$\{a_1 \cdots a_n \mid (\forall i\ a_i \in T) \wedge (S \overset{*}{\Rightarrow} a_1 \cdots a_n)\}$$

0503 Derivations

① parsing tree

② left/right-derivations and left/right-derivations have the same parsing tree

0504 Ambiguity

a grammar is ambiguous if it has more than one parse tree for some string

0601 Error Handling

① purpose of the compiler is:

a. detect non-valid program

b. translate the valid ones

② modes of error handling:

a. panic mode:

when an error is detected, discard tokens until one with a clear role is found. Continue from there.

b. error production:

specify known common mistakes in the grammar

c. error correction:

try token insertion and deletion

exhaustive search

0602 Abstract Syntax Trees (abbr. AST)

with more details added than parsing tree:

removing paretheis in trees

removing redudant nodes

...

0603 Recursive Descent Parsing (top-down parsing)

building trees according to the rules in-order using backtracking, checking

when encountering terminals.
Once matched, advance the pointer by one.

0604 Recursive Descent Algorithm (RDA)

① define some checking functions

a. a given token terminal

b. the n-th production of S

c. try all productions of S

② to start the parser

a. initialize next to point to first token

b. invoke the main parsing functions

③ Limitations

sufficient for grammars where for any non-terminal at most one production can succeed

0605 Left Recursion

① a left-recursive grammar has a non-terminal S such that:

$$S \to^{+} S\alpha \quad \text{for some } \alpha$$

and RDA will not work with left resursion problem

② in general:

$$S \to S\alpha_1 | \cdots | S\alpha_n | \beta_1 | \cdots | \beta_m$$

$$\hookrightarrow \begin{cases} S \to \beta_1 S' | \cdots | \beta_m S' \\ S' \to \alpha_1 S' | \cdots | \alpha_n S' | \varepsilon \end{cases}$$

0701 Predictive Parsing

① predictive parsers accept LL(K) grammars

- left-to-right
- left-most derivation
- K tokens lookahead

② left-factorize

add extra symbol to delay the decision to choose from 2 branches with same prefix

e.g. $E \to T+E \mid T$

$E \to TX$

$X \to +E \mid \varepsilon$

0702 First Sets

① LL1 parsing table:

$T[A,t] = \alpha$

a. if $\alpha \to^* t\beta$, then $t \in First(\alpha)$

b. if $A \to \alpha$ and $\alpha \to^* \varepsilon$ and $S \to^* \beta A t \delta$, then $t \in Follow(A)$, follow sets are always sets of terminals

② definition of First Sets:

$$First(X) = \{t \mid X \to^* t\alpha\} \cup \{\varepsilon \mid X \to^* \varepsilon\}$$

algorithm sketch for finding first sets

a. $First(t) = \{t\}$, t is terminal

b. $\varepsilon \in First(X)$

- if $X \to \varepsilon$
- if $X \to A_1 \cdots A_n$ and $\varepsilon \in First(A_i)$ for $1 \le i \le n$

c. $First(\alpha) \subseteq First(X)$ if $X \to A_1 \cdots A_n \alpha$

- and $\varepsilon \in First(A_i)$ for $1 \le i \le n$

0703 Follow Sets

① - if $X \to AB$ then $First(B) \subseteq Follow(A)$

$Follow(X) \subseteq Follow(B)$

- if $B \to^* \varepsilon$ then $Follow(X) \subseteq Follow(A)$

- if S is the start symbol then $\$ \in Follow(S)$

② algorithm sketch for finding follow sets

a. $\$ \in Follow(S)$

b. $First(\beta) - \{\varepsilon\} \subseteq Follow(X)$ for each production $A \to \alpha X \beta$

c. $Follow(A) \subseteq Follow(X)$ for each production $A \to \alpha X \beta$ where $\varepsilon \in First(\beta)$

0704 LL1 Parsing Tables

① construct a parsing table T for CFG G

for each production $A \to \alpha$ in G do:

a. for each terminal $t \in First(\alpha)$ do $T[A, t] = \alpha$

b. if $\varepsilon \in First(\alpha)$, for each $t \in Follow(A)$ do $T[A, t] = \alpha$

c. if $\varepsilon \in First(\alpha)$ and $\$ \in Follow(A)$ do $T[A, \$] = \alpha$

② if any entry is multiply defined then G is not LL(1)

0705 Bottom-Up Parsing

① a bottom-up parser traces a rightmost derivation in reverse

0706 Shift-Reduce Parsing

① idea: split string into 2 substrings
- right substring is as yet unexamined by parsing
- left substring has terminals and non-terminals
- dividing point is marked by |

② shift/move and reduce
- shift = move | one place to the right
- reduce = apply an inverse production at the right end of the left string

0801 Handles

① Want to reduce only if the result can still be reduced to the start symbol

handle formalize the above intuition: which is a reduction that also allows further reduction back to the start symbol

0802 Recognizing Handles

① Viable prefix: α is a viable prefix if there is an w such that $\alpha | w$ is a state of a shift-reduce parser

② about bottom-up parsing:
for any grammar, the set of viable prefixes is a regular language

0803 Recognizing Viable Prefixes

① rules:
a. add a dummy production $S' \to S$ to G (Grammar Rules)
b. the NFA states are the items of G including the extra production

c. for item $E \to \alpha . X \beta$ add transition

$$E \to \alpha . X \beta \xrightarrow{X} E \to \alpha X . \beta$$

d. for item $E \to \alpha . X \beta$ and production $X \to \gamma$
add $E \to \alpha . X \beta \xrightarrow{\varepsilon} X \to . \gamma$

e. every state is an accepting state

f. start state is $S' \to S$

0804 Valid Items

① item $X \to \beta . \gamma$ is valid for a viable prefix $\alpha\beta$ if $S' \to^* \alpha X w \to \alpha\beta\gamma w$

② an item I is valid for a viable prefix α if the DFA recognizing viable prefixes terminates on input α in a state s containing I

③ the item in s describe what the top of the item stack might be after reading input α

0805 SLR (simple LR) Parsing

① LR(0) Parsing: Assume
- a. stack contains α
- b. next input is t
- c. DFA on input α terminates in state s
  - Reduced by $X \to \beta$ if s contains item $X \to \beta.$
  - shift if s contains item $X \to \beta.tw$ which is equivalent to saying s has transition labeled t

then LR(0) has a <u>reduce/reduce conflict</u> if:
any state has 2 reduces items: $X \to \beta.$ and $Y \to w.$

LR(0) has a shift/reduce conflict if:
any state has a reduce item and a shift item:
$X \to \beta.$ and $Y \to w.t\delta$

② SLR:
add one condition to LR(0):
in Reduce by $X \to \beta$ if
- s contains item $X \to \beta.$
- <u>$t \in$ Follow(X)</u>

③ we can parse more grammars by using precedence declarations. e.g. '*' (multiplication) has a higher precedence than '+' (addition).

④ overall algorithm for SLR parsing

1. Let M be DFA for viable prefixes of G
2. Let $|x_1 \ldots x_n\$$ be initial configuration
3. Repeat until configuration is $S|\$$
   - Let $\alpha|\omega$ be current configuration
   - Run M on current stack α
   - If M rejects α, report parsing error
     - Stack α is not a viable prefix

   not necessarily needed
   - If M accepts α with items I, let a be next input
     - Shift if $X \to \beta.\, a\, \gamma \in I$
     - Reduce if $X \to \beta.\, \in I$ and $a \in Follow(X)$
     - Report parsing error if neither applies

0806 SLR Parsing Example

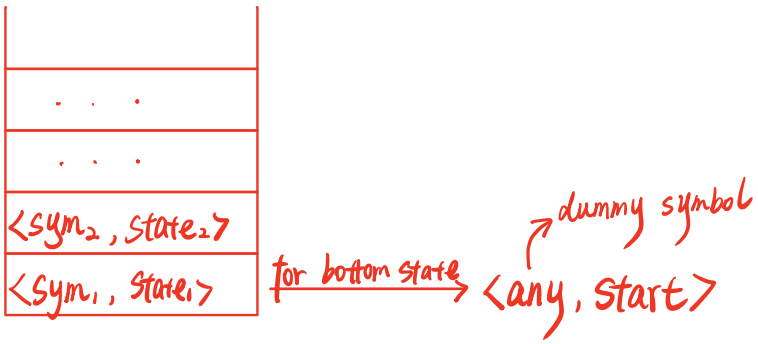
notice that after reduction, it need to restart the whole process

0807 SLR Improvements

① reduce the work of reading prefix of a stack in each step. change stack to contain pairs:

<Symbol, DFA state>

that is, the stack will be like



② goto table is just the transition function of the DFA

goto[i, A] = j if $state_i \xrightarrow{A} state_j$

③ action table

For each state s_i and terminal a

- If s_i has item $X \to \alpha.a\beta$ and goto[i,a] = j then action[i,a] = shift j
- If s_i has item $X \to \alpha.$ and $a \in Follow(X)$ and $X \neq S'$ then action[i,a] = reduce $X \to \alpha$
- If s_i has item $S' \to S.$ then action[i,$] = accept
- Otherwise, action[i,a] = error

④ SLR algorithm prseudo-code

```
Let I = w$ be initial input
Let j = 0
Let DFA state 1 have item S' → .S
Let stack = ⟨ dummy, 1 ⟩
    repeat
        case action[top_state(stack),I[j]] of
            shift k:  push ⟨ I[j++], k ⟩
            reduce X → A:
                pop |A| pairs,
                push ⟨X, goto[top_state(stack),X]⟩
            accept: halt normally
            error: halt and report error
```

⑤ we still need reserve symbols for later compiling steps

⑥ LR(1) is more powerful
build lookahead into the items.
an LR(1) item is a pair, e.g. [T→ .int*T, $]
which means T→ int*T reduce if lookahead is $

0808 SLR Examples

Converting a grammar to DFA and check whether reduce/reduce or reduce/shift conflict exist, if exist, then the grammar will not be a SLR

0901 Introduction to Semantic Analysis

① semantic analysis is the last front-end phase. lexical analysis, parsing and semantic analysis can be considered as progressive filtering

0902 Scope

① static and dynamic scoping

② a dynamically-scoped variable refers to the closest enclosing binding in the execution of the program

0903 Symbol Tables

① a symbol table is a data structure that tracks the current bindings of identifiers

② operations for simple symbol tables
   a. enter-scope
   b. find_symbol
   c. add_symbol
   e. check_scope : true if a variable already defined in current scope
   f. exit-scope

0904 Types

① a language's type system specifies which operations are valid for which type

② statically typed
   dynamically typed
   untyped : machine codes

③ type checking
   type inference : filling in missing type information

0905 Type checking

① inference rule:

$\wedge$	and
$\Rightarrow$	if-then
$x:T$	x has type T
$\vdash$	if provable, then

$$\frac{\text{hypothesis's}}{\text{conclusio}}$$

② in the type rule used for a node e:
- hypothese are the proofs of types of e's subexpressions
- conclusion is the type of node e

0906 Type environment

① a type environment gives types for free variables
- a type environment is a function from ObjectIdentifiers to Types
- a variable is free in an expression if it is not defined within the expression

② symbol O is a function from ObjectIdentifiers to Types. that is, the environment O will provide some information of types of the environment.
O can be treated to be assumption.

③ $O[T/x]$ is a function only modify one special variable
a. $O[T/x](x) = T$
b. $O[T/x](y) = O(y)$

e.g. let $x_0 : T_0$ in e_1

$$\frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \text{ in } e_1 : T_1}$$

when do type checking, x is visible in e_1
while when checking e_1, we may need to add information about x. in this case, $O[T_0/x]$ will be helpful

0907 Subtyping

① $T_1 \leq T_0$: T_1 is a subtype of T_0

② O_C all environment variables in class C

0908 Typing Methods

① in Cool, method and object identifiers live in different name spaces

0909 Implementing Type Checking

① COOL type checking can be implemented in a single traversal over the AST

a. Type environment is passed down the tree from parent to child

b. Types are passed up the tree from child to parent

1001 Static Vs. Dynamic Typing

① static type can detect common errors at compile time. however, sometimes not exactly macted variable types will also run correctly.

② dynamic type : a run-time notion
static ... : compile-time notion

③ (simple) Soundness theorem = for all expression E

$$dynamic\_type(E) = static\_type(E)$$

that is, the compiler would correctly predicate all dynamic type during execution

1002 Self Type

① Self type is a special static type.
it will replace the returning type to the current type defining the method

1003 Self Type Operations

① since dynamic_type(E) ≤ C.
which means $SELF\_TYPE_C \leq C$

a. in type checking it is always safe to replace $SELF\_TYPE_C$ by C

b. this suggests one way to handle SELF_TYPE:
replace all occurrences of $SELF\_TYPE_C$ by C

c. recall the operations on types

i. $T_1 \leq T_2$

ii. $lub(T_1, T_2)$

extend all operations for self-type

A ≤ A
C ≤ C

1. $SELF\_TYPE_C \leq SELF\_TYPE_C$
   - In Cool we never compare SELF_TYPEs coming from different classes

2. $SELF\_TYPE_C \leq T$ if $C \leq T$
   - $SELF\_TYPE_C$ can be any subtype of C
   - This includes C itself
   - Thus this is the most flexible rule we can allow

3. $T \leq SELF\_TYPE_C$ always false

   Note: $SELF\_TYPE_C$ can denote any subtype of C.

   in some very special case, it will be true, but it will be too fragile to make such a statement

4. $T \leq T'$ (according to the rules from before)

Let T and T' be any types but SELF_TYPE

1. $lub(SELF\_TYPE_C, SELF\_TYPE_C) = SELF\_TYPE_C$
2. $lub(SELF\_TYPE_C, T) = lub(C, T)$

   This is the best we can do because $SELF\_TYPE_C \leq C$

3. $lub(T, SELF\_TYPE_C) = lub(C, T)$
4. $lub(T, T')$ defined as before

1004 Self Type Usage

(go through all possible places that typeID is needed, and check whether it is allowed to use SELF_TYPE in Cool)

1005 Self Type Checking

① formal parameters cannot be SELF_TYPE
actual argument can be SELF_TYPE

② during dispatching, the method check will check the signatures of the method not in the current subtype. but it is fine, since it only checks the signature instead of contents inside the method

1006 Error Recovery

① assign type Object to ill-typed expressions
it will cause cascading errors

② introduce new type No_type
a. No_type ≤ C for all types C
b. every operation is defined for No_type with a No_type result

1101 Runtime Organization

① a. front-end phases

- lexical analysis
- parsing
- semantic analysis

} check syntax and semantics

b. back-end phases

- optimization
- code generation

② when a program is invoked:

a. OS allocates space for the program

b. code is loaded into part of the space

c. OS jumps to the entry point, i.e., "main"

1102 Activations

① an invocation of procedure P is an activation of P

the lifetime of an activation of P is

a. all steps to execute P

b. including all the steps in procedures P calls

② dynamic : lifetime
static : scope

③ 2 assumptions: (may not appliable to some languages)
a. execution is sequential: control moves from one point in a program to another in a well-defined order

b. When a procedure is called, control always returns to the point immediately after the call

④ base on assumptions:
When P calls Q, then Q returns before P returns.
↳ lifetime of procedure activations are properly nested.
which means activation lifetimes can be depicted as a tree

1103 Activation Records

① activation record (AR) / frame = the information needed to manage one procedure activation

② AR: result = return value

arguement: arguements when calling a function

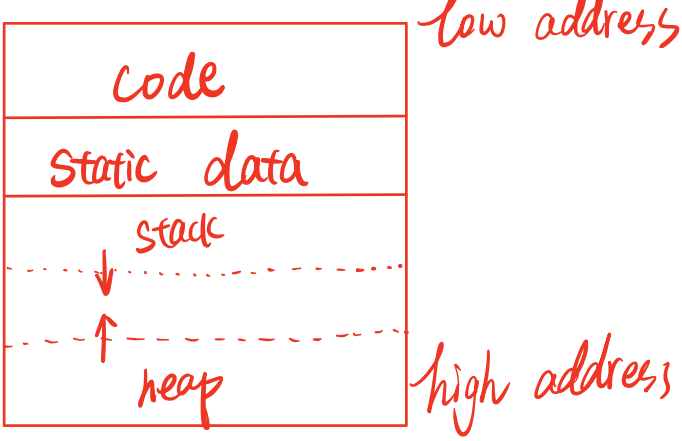
control link: pointing to the frame of a caller

return address: pointing to the address of the next command right after the callee function

1104 Globals & Heap

① globals are assigned a fixed address once. variables with fixed address are "statically allocated"

② memory layout



1105 allignment

(just talking some basic staffs about aligning data in memory)

1106 Stack Machine

① a. stack machine:

location of the operands/result is not explicitly stated which is always on the top of the stack

b. register machine:

directly store data into registers

c. n-register stack machine

keep the top n locations of the pure stack machine's stack in registers

e.g. 1-register stack machine:

the register is called the accumulator

② expression evaluation preserves the stack

1201 Introduction to Code Generation

① $a0 : 1 register holding value

$t1 : a temporary register

$sp : stack pointer

1202 Code Generation I

① A language with integers and integer operations

$$P \rightarrow D; P \mid D$$

$$D \rightarrow \text{def id(ARGS)} = E;$$

$$\text{ARGS} \rightarrow \text{id, ARGS} \mid \text{id}$$

$$E \rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4 \mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)$$

② for each expression e, we generate MIPS code:

a. compute the value of e in $a0

b. preserve $sp and the contents of the stack

③ code generation template

a. add

```
cgen(e1 + e2) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

→ code generation for e1, result in $a0

} push $a0 of e1 in stack

→ code generation for e2, result in $a0

load stored result to $t1

add $a0 and $t1

pop result stored in stack

b. substraction

```
cgen(e1 - e2) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    sub $a0 $t1 $a0
    addiu $sp $sp 4
```

c. branch

```
cgen(if e1 = e2 then e3 else e4) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    addiu $sp $sp 4
    beq $a0 $t1 true_branch
false_branch:
    cgen(e4)
    b end_if
true_branch:
    cgen(e3)
→ end_if:
```

1203 Code Generation II

① code generation template

a. function call: caller side

caller side

```
cgen(f(e1,...,en)) =
    sw $fp 0($sp)
    addiu $sp $sp -4
    cgen(en)
    sw $a0 0($sp)
    addiu $sp $sp -4
    ...
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    jal f_entry
```

b. function call: caller side

- New instruction: jr reg
  - Jump to address in register reg

```
cgen(def f(x1,...,xn) = e) =
f_entry:
    move $fp $sp
    sw $ra 0($sp)
    addiu $sp $sp -4
    cgen(e)
    lw $ra 4($sp)
    addiu $sp $sp z
    lw $fp 0($sp)
    jr $ra
```

- Note: The frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- z = 4*n + 8 [return addr. old fp]

C. variable

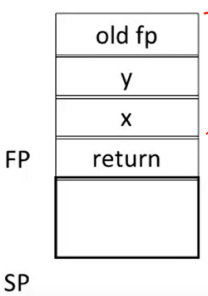
- Solution: use a frame pointer
  - Always points to the return address on the stack
  - Since it does not move it can be used to find the variables
- Let x_i be the i^{th} ($i = 1,\ldots,n$) formal parameter of the function for which code is being generated

frame pointer in callee

cgen(x_i) = lw \$a0 z(\$fp) (z = 4*i)

since caller push the variables in reverse order

Example: For a function def f(x,y) = e the activation and frame pointer are set up as follows:



caller

- X is at fp + 4
- Y is at fp + 8

callee

② production compiler

Production compilers do different things

- Emphasis is on keeping values in registers
  - Especially the current stack frame
- Intermediate results are laid out in the AR, not pushed and popped from the stack

1204 Code Generation Example

(go through a simple example of code generation using stack machine)

1205 Temporaries

① decide # temporaries involved in one function

NT(e): number of temporaries

$$NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$$
$$NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$$
$$NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$$
$$NT(id(e_1,\ldots,e_n) = \max(NT(e_1),\ldots,NT(e_n))$$
$$NT(\text{int}) = 0$$
$$NT(\text{id}) = 0$$

② layout of AR with temporaries

- For a function definition $f(x_1,\ldots,x_n) = e$ the AR has 2 + n + NT(e) elements
  - – Return address
  - – Frame pointer
  - – n arguments
  - – NT(e) locations for intermediate results

③ add a new arguement to code generation: the position of the next available temporaries

12.6 Object Layout

① objects are laid out in contiguous memory

a. each attribute stored in a fixed offset in the object

b. when a method is invoked, the object is self and fields are the object's attributes.

② object layout

- The first 3 words of Cool objects contain header information:

	Offset
Class Tag	0
Object Size	4
Dispatch Ptr	8
Attribute 1	12
Attribute 2	16
...	

- Class tag is an integer
  - Identifies class of the object
- Object size is an integer
  - Size of the object in words
- Dispatch ptr is a pointer to a table of methods
  - More later
- Attributes in subsequent slots

- Lay out in contiguous memory

③ only need to extend fields of subclass based on parent class

Observation: Given a layout for class A, a layout for subclass B can be defined by extending the layout of A with additional slots for the additional attributes of B

Leaves the layout of A unchanged

(B is an extension)

④ dynamic dispatch

- Every class has a fixed set of methods
  - including inherited methods

- A *dispatch table* indexes these methods
  - An array of method entry points
  - A method f lives at a fixed offset in the dispatch table for a class and all of its subclasses

a. we separate method calls from object layout because each object of the same class will share same methods even though they may have different values for each attributes

b. when calling a function of an object, it will bind that object to the self of the function call

1301 Semantics Overview

① a. denotational semantics: program's meaning is a mathematical function

b. axiomatic semantics: program behavior described via logical formulae

1302 Operational Semantics:

① track variables and their values with:

a. environment: mapping variables → memory location

keep track of which variables are in scope

$E[id_1 = l_1, id_2 = l_2, \cdots]$

b. store: memory locations → values

$S[l_1 \to v_1, l_2 \to v_2, \cdots]$

c. Cool object

$X(a_1 = l_1, \cdots, a_n = l_n)$

X: object; a_1: attribute; l_1: memory location

1303 Cool Semantics I

(go through semantics of some syntax)

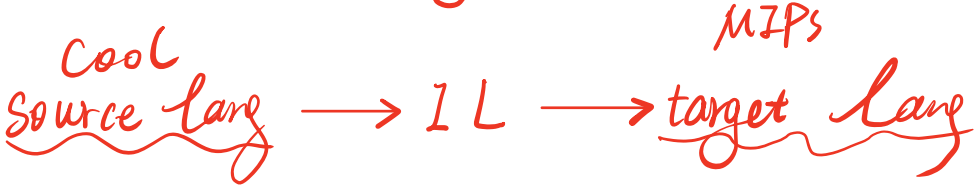
1304 Cool Semantics II

① go through semantics of class and dispatch

② operational semantics does not cover all the cases, and it is totally fine, since type checker has already dealt with that.

1401 Intermediate Code

① Intermediate code is a language between source and the target



IL has more details than source but fewer details in target

② three-address code

each instruction is of the form:

$x := y \ op \ z$

$x := op \ y$

1402 Optimization Overview

① basic block: is a maximal sequence of instructions with

a. no labels (except at first instruction)

no jumps (except in last instruction)

b. basic block is a single-entry, single-exit straight-line code segment

② a control-flow graph is a directed graph with

a. basic blocks as nodes

b. a edge from block A to block B if the execution can pass from last instruction of A to first instruction in B

1403 Local Optimization

① only focus on one basic block with no need to analyze whole procedure body

② algebaric simplification

e.g. $x := x * 0$ } delete directly since x
$x := x * 1$ } not changed

$x := x * 0 \Rightarrow x := 0$

③ constant folding

a. e.g. $x := 2+2 \Rightarrow x := 4$

b. constant folding can be dangerous:

e.g. in cross-compiler, the float point calculation result after constant folding in host machine may be different from that in target machine

c. eliminate unreachable basic blocks

d. single assignment
we can rewrite codes such that all variables are only assigned once

i. common subexpression elimination:
if 2 assignments are assigned same expression, we can replace the expression in 2nd assignment with the assigned variable in the 1st assignment.

e.g.
$$\begin{array}{l} a := e+f \\ \cdots \\ b := e+f \end{array} \Rightarrow \begin{array}{l} a := e+f \\ b := a \end{array}$$

ii. copy/constant propagation:
replace variables in later assignment with variable directly used

e.g.
```
b := z + y        b := z + y
a := b      ⇒     a := b
x := 2 * a        x := 2 * b
```

1404 Peephole Optimization

① a. Optimization can be directly applied to assembly code.

b. Peephole optimization is a short sequence of instructions

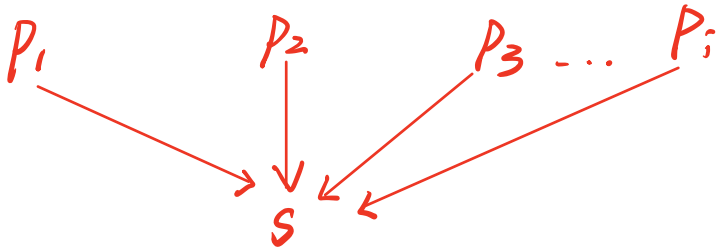
1501 Dataflow Analysis

Global optimization tasks share several traits:

- The optimization depends on knowing a property X at a particular point in program execution
- Proving X at any point requires knowledge of the entire program
- It is OK to be conservative. If the optimization requires X to be true, then want to know either
  - X is definitely true
  - Don't know if X is true
  - It is always safe to say "don't know"

1502 Constant Propagation

① symbol: $x = \begin{cases} \top & \text{value of } x \text{ not known} \\ \bot & \text{statement not executed, ie., this path would never be possible to reach} \\ c & \text{value of } x \text{ is } c \end{cases}$

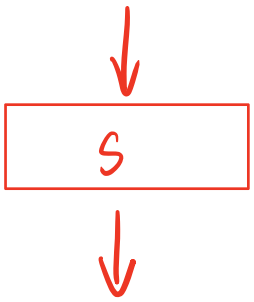


rule 1: if $C(p_i, x, out) = \top$ for any i, then $C(s, x, in) = \top$

rule 2: if $C(p_i, x, out) = c$ & $C(p_j, x, out) = d$ & $d <> c$ then $C(s, x, in) = \top$

rule 3: if $\underbrace{C(p_i, x, out) = c \text{ or } \bot}_{\text{at least one}}$ for all i, then $C(s, x, in) = c$

rule 4: if $C(p_i, x, out) = \bot$ for all i, then $C(s, x, in) = \bot$



rule 5 (with lowese priority)

if $C(s, x, in) = \perp$,

then $C(s, x, out) = \perp$

rule 6

if c is a constant

$C(x := c, x, out) = c$

rule 7

$C(x := f(\cdots), x, out) = \top$

↳ complicate expression for x

rule 8

if $x <> y$, where y is assignment in S

$C(y := \cdots, x, out) = C(y := \cdots, x, in)$

② algorithm

1. For every entry s to the program, set C(s, x, in) = ⊤
2. Set C(s, x, in) = C(s, x, out) = ⊥ everywhere else
3. Repeat until all points satisfy 1-8:
   Pick s not satisfying 1-8 and update using the appropriate rule

1503 Analysis of Loops

① to avoid entering into a infinite loop during analysis, we need to set an initial value ⊥ to break the loop.

1504 Ordering

① ordering of abstract value

$$\bot < c < \top$$

② lub operation on abstract value

then rules 1-4 (in 1502) can represented

as

$$C(s, x, in) = lub\{C(p, x, out) \mid p \text{ is a predecessor of } s\}$$

③ ordering explains why constance propagation terminates, since lub operation only inscrease at most to T

then

Number of steps (for constance propagation)

= Number of C(...) values computed * 2

= Number of program statements * 4

15.05 Liveness Analysis

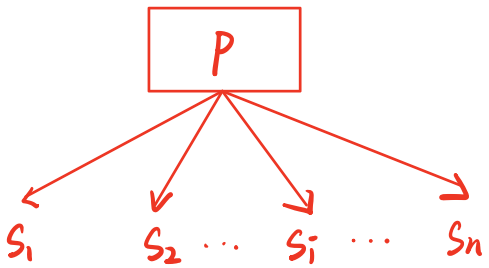
① Live means may be used in the future

② A variable x is live at statement s if

- There exists a statement s' that uses x
- There is a path from s to s'
- That path has no intervening assignment to x

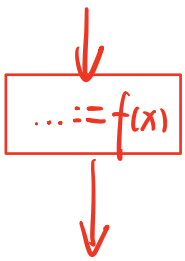
③

rule 1:



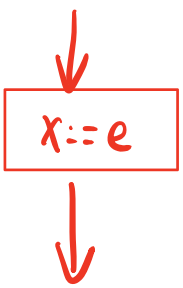
$L(p, x, out) = \vee\{L(s, x, in) \mid s \text{ a successor of } p\}$

rule 2:



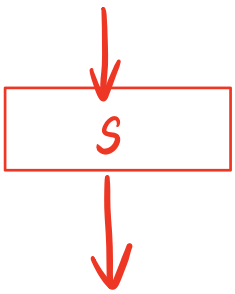
$L(s, x, in) = true$ if s refers to x on rhs

rule 3:



$L(x := e, x, in) = false$ if e does not refer to x

rule 4:



$L(s, x, in) = L(s, x, out)$ if s does not refer to x

② algorithm

1. Let all L(...) = false initially

2. Repeat until all statements s satisfy rules 1-4

   Pick s where one of 1-4 does not hold and update using the appropriate rule

160l Register Allocation

① rewrite intermediate code to use no more temporaries than there are machine registers

method:

- assign multiple temporaries to each register
- without changing problem behavior

② *Temporaries t_1 and t_2 can share the same register if at any point in the program at most one of t_1 or t_2 is live.*

Or

If t_1 and t_2 are live at the same time, they cannot share a register

③ register interference graph (RIG)

a. a node for each temporary

b. an edge between t_1 and t_2 if they are live simultaneously at some point in the program

1602 Graph Coloring

① graph coloring is NP-hard problem

use heuristic method:

Observation:

- Pick a node t with fewer than k neighbors in RIG
- Eliminate t and its edges from RIG
- If resulting graph is k-colorable, then so is the original graph

algorithm:

Graph Coloring

1. The following works well in practice:
   - Pick a node t with fewer than k neighbors
   - Put t on a stack and remove it from the RIG
   - Repeat until the graph is empty
2. Assign colors to nodes on the stack
   - Start with the last node added
   - At each step pick a color different from those assigned to already colored neighbors

1603 Spilling

① when graph-coloring fails with given number of registers in machine, we need to spill some registers, i.e., store them somewhere in memory stack

steps:

when we can not choose any nodes with less than k edges (k is number of registers in machine)

a. random pick one node N. and

remove it from graph. and continue do graph-coloring for rest of the nodes

b. after all nodes are removed from graph and pop nodes from stack.

when poping N, we can check whether it can be properly colored, which is called optimistic coloring

if optimistic coloring fails, we need to allocate a memory location for f, we call the address fa.

then we need insert codes into original codes:

- Before each operation that reads f, insert
  f := load fa
- After each operation that writes f, insert
  store f, fa

② when deciding which node to do spilling

Possible heuristics:

- Spill temporaries with most conflicts
- Spill temporaries with few definitions and uses
- Avoid spilling in inner loops

1604 Managing Caches

(elaborate cache managing by showing an example of loop interchanging)

1701 Automatic Memory Management

① an object x is reachable if and only if:

a. a register contains a pointer to x, or

b. another reachable object y contains a pointer to x

② reachability is an approximation:

an object allocated but not reference to will never be able to be used again.

However, an object is being refercenced does not gurantee it is always needed in future, since we can assign value to a variable but never use that variable.

Thus, reachability is an approximation.

③ algorithm for garbage collection in Cool

- Every garbage collection scheme has the following steps
  1. Allocate space as needed for new objects
  2. When space runs out:
     a) Compute what objects might be used again (generally by tracing objects reachable from a set of "root" registers)
     b) Free the space used by objects not found in (a)

- Some strategies perform garbage collection before the space actually runs out

1702 Mark and Sweep

① mark phase:

mark all reachable object with 1 in the special bit

```
// mark phase

let todo = { all roots }
while todo ≠ ∅ do
    pick v ∈ todo
    todo ← todo - { v }
    if mark(v) = 0 then      // v is unmarked yet
        mark(v) ← 1
        let v1,...,vn be the pointers contained in v
        todo ← todo ∪ {v1,...,vn}
    fi
od
```

② Sweep phase

- The sweep phase scans the heap looking for objects with mark bit 0
  - these objects were not visited in the mark phase
  - they are garbage
- Any such object is added to the free list
- Objects with a mark bit 1 have their mark bit reset to 0

```
// sweep phase
// sizeof(p) is the size of block starting at p

p ← bottom of heap
while p < top of heap do
  if mark(p) = 1 then
     mark(p) ← 0
  else
     add block p...(p+sizeof(p)-1) to freelist
  fi
  p ← p + sizeof(p)
od
```

③ Space for todo list when mark phase:

when we do GC, we usually run out of heap. thus, we may not be able to allocate enough space to store all nodes for todo list in heap directly.

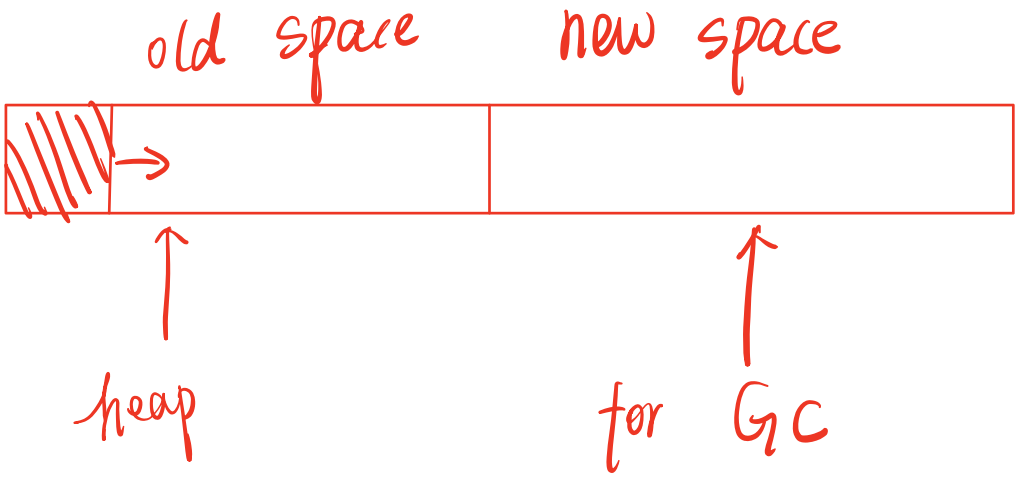
then we can use reverse pointer to handle this

④ a. Mark and Sweep do not move position of pointers

b. mark and sweep can fragment memory after sweeping. thus, it is necessary to merge free space into one big block

1703 Stop and Copy

① reserve half of the space for GC



When old space is full, GC would copy reachable objects in heap into new space, and switch roles of old and new space

② update value of pointers in new copied area

③ complete algorithm for Stop and copy

```
while scan <> alloc do
    let O be the object at scan pointer
    for each pointer p contained in O do
        find O' that p points to
        if O' is without a forwarding pointer
            copy O' to new space (update alloc pointer)
            set a word of old O' to point to the new copy
            change p to point to the new copy of O'
        else
            set p in O equal to the forwarding pointer
        fi
    end for
    increment scan pointer to the next object
od
```

17.04 Conservative Collection

① treat all data looks like a memory address as pointer

a. it must be aligned

b. it must point to a valid address in data segment

② we can not move an object. if we do so, we may change the value of something not like integer (e.g. integer)

1705 Reference Counting

① using a counter to record the number of being referenced. when the counter goes to 0, the object will become reachable and be be freed

1801 Java

(talking about history of Java)

1802 Java Arrays

① subtyping rule for Java Arrays:

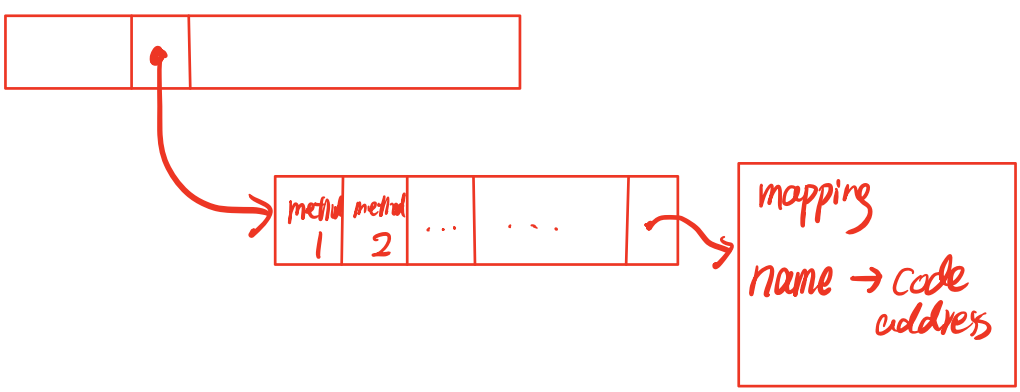
A[] < B[] if A = B

1803 Java Exception

① exception happened in object finalization if it is not handled by itself in that finalization would simply be ignored

1804 Java Interface

① Since Java class could implements multiple interfaces, it is not possible to guarantee methods are put in a fixed offset.



18.05 Java Coercions

1. a coercion is just a primitive function the compiler inserts for programmer
2. types of coercions/casts

   widening = always succeed (int → float)
   narrowing = may fail if data can't be converted to desired type (float → int, downcasts)

3. coercion could lead to surprising result

18.06 Java Threads

(talking about thread features of Java)

1807 Other Topics

① a class is initialized when a symbol in the class is first used instead of loading class.
→ this makes initialization error predicable. If we do so on class loading, it will become somehow random since when a class loading is hard to predicate.

② brief steps of how to initialize a class

1. Lock the class object for the class
   - Wait on the lock if another thread has locked it
2. If the same thread is already initializing this class, release lock and return
3. If class already initialized, return normally
4. Otherwise, mark initialization as in progress by this thread and unlock class

5. Initialize superclass, fields (in textual order)
    - But initialize static, final fields first
    - Give every field a default value before initialization
6. Any errors result in an incorrectly initialized class, mark class as erroneous
7. If no errors, lock class, label class as initialized, notify threads waiting on class object, unlock class