Intro to Compilers

What is a compiler?

- A program that translates software written in one language into another language.
- Should improve the program in some way
- Ex: C , C++ are both compiled

What is an interpreter?

- A program that reads an executable program and produces the results of executing that program
- Ex: Python, Scheme are both interpreted.

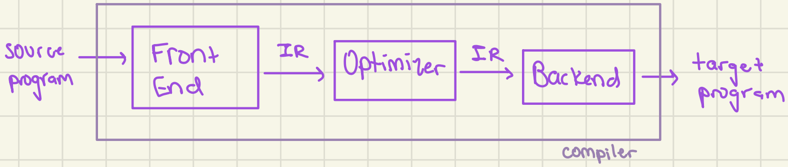
Is Java interpreted or compiled?

- Both! Kinda
- Java is compiled into bytecode (code for the JVM)
- the bytecode is then interpreted
- some bytecode may be compiled (just-in-time compilation)

What are the principles of compilation?

- Compiler must preserve the meaning of the input prog
- Compiler must discernibly improve the input prog

What is the structure of a compiler?



What does the Front End component do?

- Deals with the source code. Takes a stream of chars & converts into a stream of classified code.
- INPUT: source-language program
- Performs 3 types of tasks/operations:
  1) Lexing (scanning): stream of characters → stream of words (e.g., removing whitespace)
  2) Parsing: is stream of words a "sentence" in the source language?
  3) Static semantic analysis: Is the sentence (statically) meaningful?
- OUTPUT: Intermediate representation (IR)

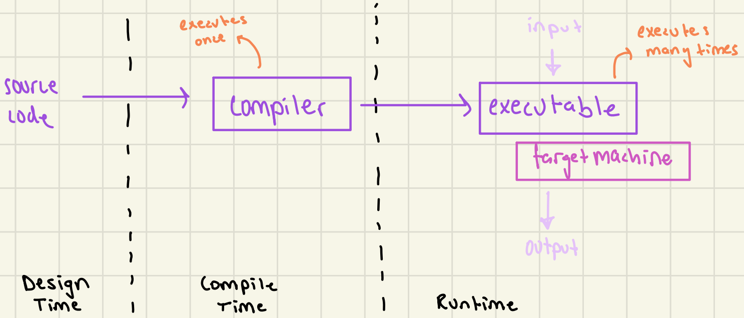
What does the Optimizer do?

- INPUT: IR
- Focuses on improving program efficiency. multiple rewrite passes; reducing code size
- OUTPUT: Intermediate representation (IR)

What does the Backend Component do?

- Deals with the target machine
- INPUT: Intermediate representation (IR)
- Instruction selector: turns IR into target machine instructions
- Register allocator: Fit the finite register set given to you by target machine, to the program that you have.
- Instruction scheduler: reorder instructions for speed
- OUTPUT: target-language program

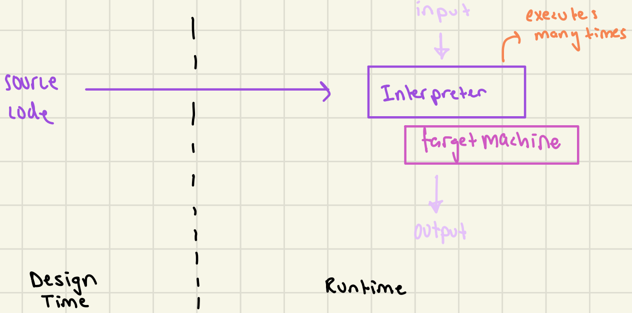
What is the timing breakdown for compiled languages?



What happens at each stage?

- → Design Time: design & implementation of the compiler
- → Build Time: using a compiler to compile the compiler
- → Compile Time: execute the compiler to translate source-language program to target-machine code
- → Runtime: target-machine code executes
- → Just-in-Time compiler (JIT): a runtime compile-time

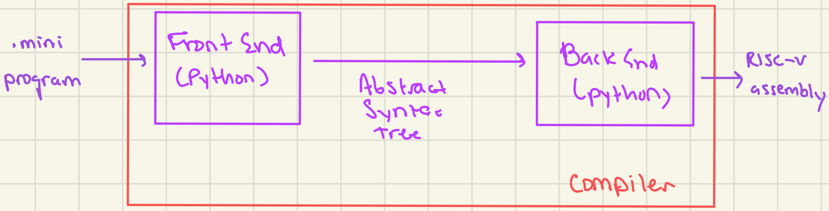
What is the timing breakdown for interpreted languages?



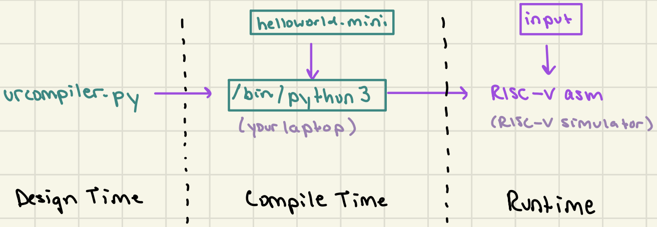
So, why study compilers?

- → Fundamental in computation
  - key drivers of performance
  - enable use of programming langs
- → Broad applications
  - * classic compilation (e.g. gcc, rust) *embedded langs (e.g., macros in excel)
  - * scanning & parsing (e.g. HTML, SAT queries) *domain-specific langs
- → Fundamental in CS
  - computer organization/architecture, assembly programming, OS, algorithms, formal languages
- → Intrinsically interesting!

What will my compiler look like?



Timing for my compiler?



Front-End: Scanners

Ch.2

RECALL: What are the 3 main tasks of the Compiler FrontEnd?

1. Lexical Analysis (Scanning):
   - → concerned with microsyntax
   - → maps: stream of chars ⤳ stream of words
2. Parser:
   - → concerned with grammar
   - → maps: stream of words → annotated sentences
3. Semantic Elaboration
   - → checks that sentences are meaningful

Example?

→ Say that source lang is English.

"Compilers arp fun" → error caught by scanner b/c "arp" isn't a valid word.

"Compilers are fun" → All legal words; NOT caught by scanner

↳ Grammatically incorrect; caught by **parser**

"Compilers walk fun" → All legal words; NOT caught by scanner

↳ Grammatically correct; NOT caught by **parser**

↳ Semantically non-meaningful: caught by **semantic analysis**

What does the Scanner do?

→ The only part of the compiler that **reads every character**

→ Maps streams of chars to streams of tokens.

- Token: <syntactic category, lexeme>
- e.g.: INPUT: "Compilers are Fun."

  OUTPUT:

  stream of tokens

  <NOUN, "Compilers">, <VERB, "are">, <ADJ, "Fun"> <ENDMARK, ".">

→ Finds microsyntax errors: unrecognized lexemes, spelling mistakes

- microsyntax: specifies correct spelling of all words in the lang.

What is the difference between Scanning and Parsing?

→ Scanning: identifies & classifies words, recognizing words in the lang.

→ Parsing: identifies sentences

- grammar: specifies all legal sentences in the language

Why separate the two?

→ greater efficiency; parsing is harder than scanning; better to give the parser only well-formed tokens

→ Scannerless parsing – e.g. combining the 2 – is possible.

→ Scanner Construction is based on Finite Automata

What is the plan for lexical analysis (e.g. the compiler's plan)?

Design Time	Build Time	Build Time	Runtime
• specify microsyntax using REs	• Convert RE to FA	• Generate code to implement FA	• Code analyzes input to produce <category, lexyme> pairs

– Recognizing Words: Finite Automata –

RECALL: What is a DFA?

→ A deterministic finite automaton (DFA) is a 5-tuple ($S, \Sigma, \delta, s_0, S_A$), where

1. S is a finite set, called the states
2. Σ is a finite set, called the alphabet
   - represents the "input alphabet"
   - indicates the allowed input symbols
   - for ex, with binary string inputs, $\Sigma = \{0,1\}$
3. $\delta : S \times \Sigma \longrightarrow S$ is the transition function
4. $s_0 \in S$ is the start state
5. $S_A \subseteq S$ is the set of accept states.

→ A DFA accepts a string iff:

- Starting in s_0 & consuming each char in the string according to the transition function, leaves it in an accepting state.

→ We regard **Recognizers** as DFAs

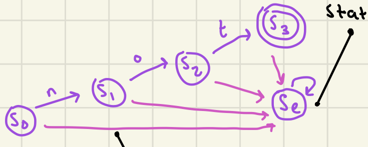
Example of a Recognizer?

→ Recognizing the word "not":

```
c ← next char
if c = 'n' then {
    c ← next char
    if c = 'o' then {
        c ← next char
        if c = 't'
            then return <NOT, "not">
            else report error
    } else report error
} else report error
```

Code

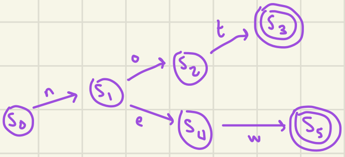
Recognizer



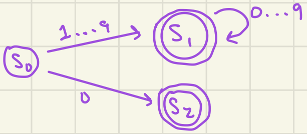
se = error state

all states have an implicit edge to the error state

→ Ex2: Recognizing "not" or "new"



How would you recognize the infinite set of integers?



What are some key terms & Facts regarding Finite Automata (FA)s?

- → language: set of words accepted by an FA. Countable set of strings over a fixed alphabet
- → L(A): lang. recognized by an F.A. A
- → An FA recognize exactly one language
- → The transition diagram of FA specifies how to spell every word in L(FA)
- → 2 languages cannot be recognized by the same FA or the same RE.

— Specifying Languages: Regular Expressions —

Why use REs?

- → FAs are precise but not concise. REs are precise & concise
- → Set of langs that can be recognized by an FA = " by an RE

RECALL: what are the rules for REs?

RE Syntax?

- → ε is an RE, and $L(\varepsilon) = \{\varepsilon\}$ (empty string)
- → if $a \in \Sigma$, then a is an RE and $L(a) = \{a\}$
- → Let $X = \{a, b\}$ and $Y = \{c, d\}$

1. UNION: $X \mid Y = X \cup Y = \{a, b, c, d\}$
2. CONCATENATION: $X \circ Y = \{xy \mid x \in X \text{ and } y \in Y\} = \{ac, ad, bc, bd\}$
3. KLEENE CLOSURE: $X^* = \{X_1 X_2 \ldots X_k \mid k \geq 0 \text{ and each } x_i \in X\}$
   $= \{\varepsilon, a, b, ab, ba, aab, abba, ba, \ldots\}$

What is some syntactic sugar for REs?

- → Positive closure: r^+ (equivalent to rr^*). Specifies "1 or more rs"
- → $[0...9] = 0|1|2|3|4|5|6|7|8|9$
- → $[a...z] = a|b|c|...|z$
- → Complement: $\wedge c = \Sigma \setminus \{c\}$

What are the steps for building a Scanner?

1. Build REs For each category in the programming language.
   - → e.g., cat1 = integers (like $[0...9]^*$ or smthn)
   - cat2 = keywords (if|while|then|for|...)
   - cat3 = identifiers $(a\text{-}z|A\text{-}Z)(a\text{-}z|A\text{-}Z|0\text{-}9)^*$
2. Combine into single RE (union) (cat1|cat2|cat3)
3. Convert RE ⟶ NFA using Thompson's Construction (1)
4. Convert NFA ⟶ DFA using subset construction (2)
5. DFA minimization: DFA ⟶ DFA (3)
6. Scanner Generator: DFA ⟶ code (4)
   - skeleton scanner + generated table
   - generated scanner

– Converting REs to Finite Automata – (1)

RECALL: What is an NFA?

→ in nondeterminstic finite automatons (NFAs), several choices may exist for the next state at any point.

A nondeterministic finite automaton is a 5-tuple $(S, \Sigma, \delta, s_0, S_A)$, where

1. S is a finite set of states
2. Σ is a finite alphabet
3. $\delta : S \times \Sigma_\varepsilon \longrightarrow P(S)$ is the transition function
4. $s_0 \in S$ is the start state
5. $S_A \subseteq S$ is the set of accept states.

→ $\delta : S \times \Sigma_\varepsilon \longrightarrow P(S)$, where δ takes an input of the cartesian set of all possible combos of states (S) and input symbols plus the empty string (Σ_ε), and produces the power set of S, aka the set of all possible next states.

→ $P(Q)$ = power set of Q = collection of all possible subsets of Q.

- ex: $Q = \{1, 2\}$ ... $P(Q) = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$

so How do NFAs compute?

→ When the NFA arrives at a state with multiple ways to proceed (like if we were at q_1 and the next input symbol is a 1, we can either stay in q_1 or move to q_2), the machine splits into multiple copies of itself and then follows all of the possibilities in parallel.

- each copy of the machine takes one of the possible paths & then continues on reading the input
- Every time there are choices, the machine "splits" again

→ NFAs are like a parallel computation where multiple independent "threads" can be running concurrently.

What happens when the NFA arrives on a state that has ε on an exit arrow?

→ Similar: without/before reading any further input, the machine splits into at least two but possibly more copies — one that stays at the current state, and one following each of the exiting ε-labeled arrows.

- For ex, when it arrives at q_2 and the next symbol is a 0, N_1 splits into 2 copies befor reading the next symbol — one that stays at q_2 and one that advances to q_3

What are the 2 types of NFA computation models?

→ Omniscient NFA: at each nondeterministic choice, follow the transition that leads to an accepting state for the input string

→ Cloning NFA: the one described above (from COMP 455)

What is Thompson's Construction?

→ Method to generate an NFA from an RE

→ For each symbol in Σ and each R.E. operator, there is an **NFA template**

1) Join templates together to form larger REs

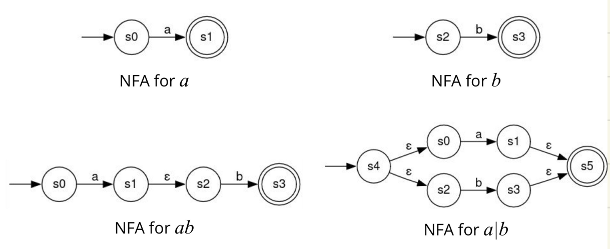
- Join with ε transitions
- Join in precedence order

2) Adjust set of accepting states

What is the precedence order?

( ) , * , concatenation , union —left-associative

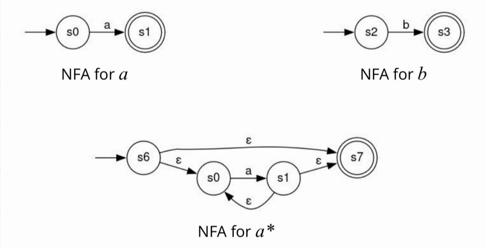
What are the templates?


NFA for a

NFA for b

NFA for ab

NFA for $a|b$


NFA for a

NFA for b

NFA for $a*$

Other properties?

→ Single start & single accepting state

→ No transitions return to start state

→ No transitions leave accepting state

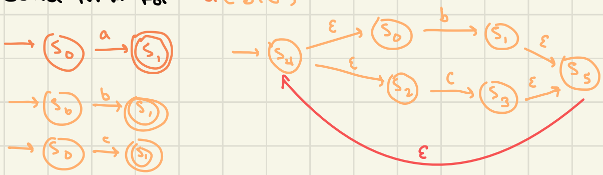
→ Each state has max 2 transitions per symbol; one exiting and one entering

→ Each state has max 4 ε-transitions; 2 exiting & 2 entering

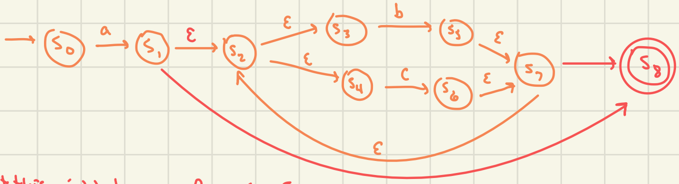
Example of using Thompson's Construction?

→ build NFA for a(b|c)*

1)



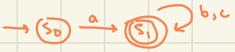
2)



(wait this might be wrong? version from textbook on next page)
*have to check these with prof

Why use Thompson's construction if we can just make a minimal DFA?

→ e.g., for $a(b|c)^*$:



→ ANS: b/c it can't be auto-generated ; we had to come up with it in our brains.

→ Thompsons Construction process can be coded / automated

— Subset Construction : NFA ⟶ DFA — (2)

What do we do in this step?

→ Convert the NFA into a DFA

→ Each state in DFA = set of states in NFA

→ Transitions in DFA represent the possible transitions from the set of NFA states.

→ Subset construction : the alg to convert NFA → DFA

How do we execute subset construction?

1) start state q_0 in DFA = set of NFA states including :

- n0, NFA start state
- all states reachable from n0 by following ε
- $q_0 = s_0$

2) Given state q in DFA :

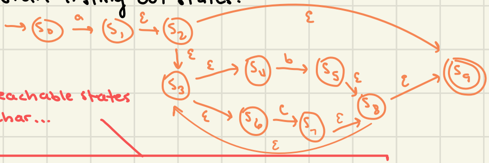
- For each NFA state n in q, for each char c : follow all possible NFA transitions, including ε
- the set of reachable states from doing this = new q'
- add q' and transition to DFA

3) iterate step 2 until no more states are added

4) accepting states in DFA : contains an NFA accepting state

→ So basically, make a table & start listing out states.

→ Ex: NFA for $a(b|c)^*$:



Set of reachable states for char...

DFA State Name	NFA States	a	b	c
q_0	{s0}	{s1, s2, s9, s3, s4, s6, }	-none-	-none-
q_1	{s1, s2, s9, s3, s4, s6, }	-none-	{s3, s9, s4, s6, s5, s8}	{s3, s9, s4, s6, s7, s8}
q_2	{s3, s9, s4, s6, s5, s8}	-none-	q_2	q_3
q_3	{s3, s9, s4, s6, s7, s8}	- none -	q_2	q_3

this becomes our next state q1!

if you write out the states, it ends up being = to set of NFA states listed for q2!

why not including s2 in the set for "b" in row for q1?

What is the pseudocode for subset construction?

→ When alg halts, each $q_i \in Q$ corresponds to a state $d_i \in D$ in the DFA

→ Alg builds elements of Q by following the transitions that the NFA can make on a given input.

→ each element of Q is a subset of N (the set of states in the NFA)

```
q0 = follow_epsilon ( {s0} )
Q = {q}
workList= {q0}
while (workList != { }):
    workList. remove (q)
    for char in Σ :
        tmp= follow_epsilon (delta (q,c))
        if tmp != { } :
            if tmp not in Q :
                Q . append ({tmp})
                workList.append({tmp})
            T[q, char] = tmp
```

each q represents a valid config of the original NFA

helper function that applies the NFA's transition function to each element of q, and returns $\bigcup_{s \in q_i} \delta_N(s,c)$

What is follow_epsilon?

→ helper function that takes a set S of NFA states. Then, for each state $s_i \in S$, it adds to S any NFA states reachable by following one or more ε-transitions from s_i. It returns the "updated" version of S, with all of those states added to it.

What type of algorithm is this?

→ Fixed-point computation : guaranteed to terminate

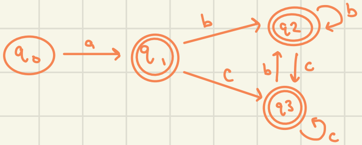
- Monotonic construction of finite set (Q)
- exists a finite # of possible states to add.
- Halts when it stops adding to the set.

How do we construct the DFA after generating subsets & DFA states?

→ We now have our set of DFA states Q, and their transitions:

DFA State	NFA States	a	b	c
q_0	{s0}	q1	-none-	-none-
q_1	{s1, s2, s9, s3, s4, s6, }	-none-	q2	q3
q_2	{s3, s9, s4, s6, s5, s8}	-none-	q_2	q_3
q_3	{s3, s9, s4, s6, s7, s8}	- none -	q_2	q_3

→ q_0 becomes start state; any $q_i \in Q$ who contains a state $\{n_i \in N \text{ (NFA)} \mid n_i = \text{accepting state}\}$, becomes an accepting state.



— DFA Minimization: Hopcroft's Algorithm —

What is a partition of a set?

→ For a set Q: a collection of subsets of Q which are disjoint & whose union gives you back the original set. Basically dividing elements of Q up into groups.

What is Hopcroft's Algorithm?

→ Idea: detect when 2 states produce the same behavior on any input string

1) Build a partition of DFA states, $P = \{p_1, p_2, \dots p_n\}$ s.t.:

- For all c in Σ: if x_a, x_b in P_s, $\delta(x_a, c) = x_i$, and $\delta(x_b, c) = x_j$, then x_i, x_j must be in the same partition.

  (basically, if P1 has a and b, and char c transitions a & b to c & d, respectively — THEN, c and d must be in the same partition.)

2) Start with the initial partition $P_0: \{\overset{\text{accepting state}}{S_A}, \{S - S_A\}\}$

3) Split partitions to satisfy the requirements

4) Iterate until reqs satisfied. Iterate for each $c \in \Sigma$

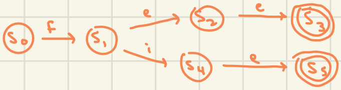
5) Create minimized DFA with 1 state for each partition group.

Example?

→ Final req:

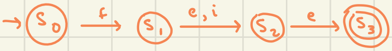
- 2 DFA states $a, b \in P_s$ have the same behavior in response to all input chars.
- Property holds for every $p_s \in P$, every pair of states $a, b \in p_s$, and every input char c.
- When examining a partition $\overset{P_x}{\{a,b,c\}}$, if on a given char, all states transition to a state inside P_x ... the states can remain in this group

→ Non-minimized DFA recognizing fee | fie:



Step	Current Partition	char	set	action
0	{{s0,s1,s2,s4},{s3,s5}}	-	-	-
	{{s0,s1,s2,s4},{s3,s5}}	e	{s0,s1,s2,s4}	Split off {2,4} why? because on char "e", s2 & s4 both transition to states in the partion {s3,s6}. So they must be in the same partition.
	{{s2,s4}{s3,s5},{s0,s1}}	f	{s0,s1}	split off {s0}
	{{s2,s4},{s3,s5},{s0},{s1}}			

→ Minimized DFA:



– Scanner Generator: Generate Scanner From DFA –

How do we generate a scanner from a DFA?

→ Skeleton scanner + DFA tables

→ Direct-coded scanner

→ Handwritten scanner

What is a Skeleton Scanner?

→ Scanner generator: RE → DFA, DFA → table

→ Skeleton Scanner interprets the tables to simulate the DFA

→ Generated Scanner uses the same skeleton for every RE

→ Tables encode specifics of given RE

Example?

→ Skeleton Scanner:

```
char = next_char
state = s_0
while (char ≠ EOF) :
        state = δ(state, char)
        char = next_char
if (state = final state):
        //report success
else:
        report failure
```

→ Table:

δ	a	b	other
s0	s1	s0	se
s1	s0	s1	se
se	se	se	se

What is a Direct-Coded Scanner?

→ Scanner Generator: RE → DFA, DFA → code

→ Transitions are compiled into conditional logic

→ Generated Scanner is different for each RE

→ Generated Code encodes specifics of RE

→ Low overhead per char

High level: How is the scanner generated?

→ Build REs for each category in programming language

→ Combine into a single RE (union)

cat1 | cat2 | cat3

- Ex: cat1 = (if|while|for|then) cat2 = $(a\text{-}z|A\text{-}Z|0\text{-}9)^*$

→ Use associated DFA to build scanner

Recap: How does the scanner work?

→ Reads enough input to recognize a word – DFAs read all the input

→ Reads until it reaches state s, next char c

- if $\delta(s, c) = s_e$ and s in S_A... the scanner found a word
- else, scanner backs up until it finds an accepting state or it exhausts the current lexeme.

→ Returns a token <lexeme, category>

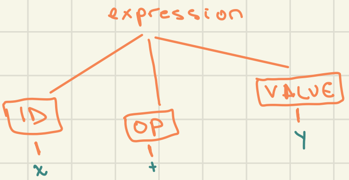
Summary / Recap?

→ FAs are good for recognizing words

→ REs are good for specifying the set of words to recognize

What is an Abstract Syntax Tree (AST)?

→ Represents the structure of a languege, as a tree

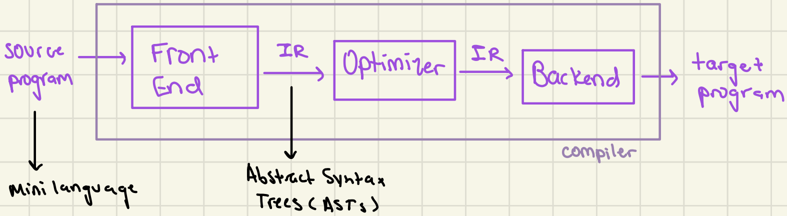


→ the structure that shows up here, is given by your parser.

→ parser generates a parsetree, which has tokens & stuff & isn't the same as an AST

→ Lab 0: Visit AST & pretty print an expression

The Mini Compiler: Front End



→ Scanning : Mini.g4 grammar ⟶ generated parse tree (ANTLR4)

→ Parsing : generated parse tree ⟶ AST

What is an Abstract Syntax Tree?

→ A graph that describes the structure of your input source code.

→ AST design is closely linked with that of a compiler

Example?

→ Construct the syntax tree for 5 + (4 * 3) $

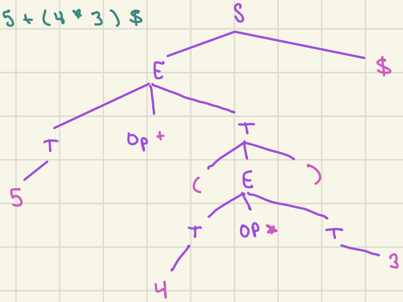
S ::= E $

E ::= T (Op T)*

T ::= (E) | num

Op ::= + | *

("::=" means "is defined as")



→ Precedence rules (eg. order of ops) must be enforced for the correct AST to be generated.

What is the Visitor Model/ Design Pattern?

→ We need to be able to traverse our ASTs

→ MiniToASTVisitor handles construction of the AST. It builds AST from the parse tree.

→ Subsequent passes implement the ASTVisitor to traverse the AST constructed.

What does ASTVisitor provide?

→ abstract base classes with visit methods for all AST node types.

→ Establishes a concrete visitor pattern interface (All visitors must implement)

Front End: Parsers

Ch. 3

RECAP: Where are we?

1. Lexical Analysis (Scanning): ✓
   - → concerned with microsyntax
   - → maps: stream of chars ⟿ stream of words
2. Parser:
   - → concerned with grammar
   - → maps: stream of words ⟶ annotated sentences
   - → looks at every word
3. Semantic Elaboration
   - → checks that sentences are meaningful

Front End

What does the parser do?

→ Given an input of **tokens** <category, lexeme>, maps it into a sentence.

→ Ex: int x; ⟶ <type, "int"> <ID, "x">

What is parsing?

→ Given a stream of words, s, and a grammar G, find a **derivation** in G that produces S.

→ Goal: recognize syntactically valid sentences in the language.

1. Specify syntax of the lang. using grammar G.
2. Test membership in L(G)

What are the 2 methods for parsing?

→ Top-Down parsing: Using G, try to recreate the sentence s. Start with G.

- begin with root node & build down
- root node ≈ grammar's start symbol

→ Bottom-Up parsing: Starting with s, identify rules to verify that it fits in G.

- begin with leaf nodes & build up
- leaf nodes ≈ string of terminals

How do we specify the syntax of a lang?

RECALL: What is a CFG?

→ Context Free Grammars! COMP 455!!

A context-free grammar is a 4-tuple (V, T, P, S), where

1. V is a finite set of syntactic **variables**, aka nonterminals
2. T is a finite set, disjoint from V, called the **terminals**,
   - disjoint = no common elements between V and Σ. T = Σ
3. P is a finite set of **rules**, with each rule being: a variable and a string of variables & terminals, and
4. S∈V is the start variable.

→ The language of a given CFG = set of sentences that can be derived from S.

Example?

→ Grammar G1: S ⟶ aSb

| SS

| ε

Rules

V → Variables = {S}

T → Terminals = {a, b, ε}

S → Start variable = S

→ Ex strings in G_1: ε, aSb, aaSbb, aaaSbbb, aSSb, ...

RECALL: What is a derivation?

→ A sequence of applications of rules to transform the start symbol in the grammar, into a sentence in the language.

What is the difference between Context-Free & Regular languages?

→ Regular Languages: (e.g., $a^* b^*$)

- cannot count to an unbounded number
- recognized by FAs

→ CF Languages: (e.g., $a^n b^n$)

- can count, but cannot recognize context
- recognized by nondeterministic PDAs

→ (set of all Reg. Langs) ⊆ (set of all CF langs)

What are the key components of a derivation?

→ STEPS to derive:

$$\text{Start} \rightarrow \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_{n-1} \rightarrow \gamma_n \rightarrow \text{sentence}$$

1) select a non-terminal $A \in \gamma_i$

2) Use rule $A \rightarrow \beta$ to expand A to get γ_{i+1} (replace A with β)

→ If γ_i contains only terminal symbols, it is a sentence in $L(G)$

→ If γ_i contains 1 or more non-terminals, it is a sentential form.

→ Parsing = finding a derivation for a sentence

What is a Sentential Form?

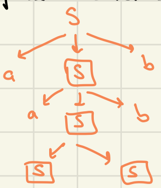
→ A sequence of symbols that occurs at one step in a valid derivation.

What is a parse tree?

→ A diagram that illustrates how a string gets generated. Alternative to derivations, e.g. another way to represent a sentence

Example?

→ From G_1 on prev page ... tree for string aaSSbb



What is an Ambigous grammar?

→ A grammar that contains at least one sentence which can be derived in multiple ways via multiple parse trees.

- If the same string can be made from 2 diff parse trees, the grammar is ambigous
- If same string can be made from 2 diff derivations, doesn't mean anything

Formal definition?

→ A string w is derived ambiguously in CFG G if it has 2 or more leftmost derivations.

→ A grammar G is ambiguous if it generates some string ambiguously.

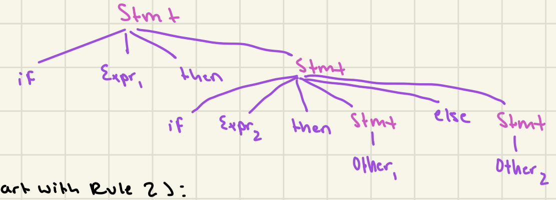
Example of an Ambiguous Grammar?

→ Grammar G_2 :

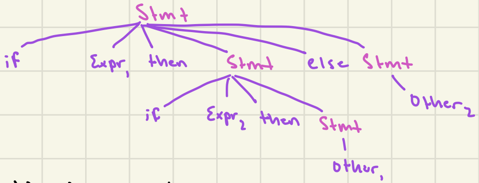
Stmt → if Expr then Stmt
| if Expr then Stmt else Stmt
| Other

→ Ex: deriving sentence S : if $Expr_1$ then if $Expr_2$ then $Other_1$ else $Other_2$

→ PARSE TREE 1 :



→ PARSE TREE 2 (Start with Rule 2):



What is the difference between Parse Trees & Derivations?

→ Derivations: capture the order in which rules are expanded ; 2 diff derivations may prodvce the same parse tree

→ Parse Trees : capture the structure & meaning of the sentence

- For ambiguity, we are concerned with structure & meaning

→ To focus on parse trees, we fix the derivation order

What are the derivation orders?

→ **Leftmost Derivation**: replace the leftmost variable at each step

- A left-setential form occurs in a leftmost derivation

→ **Rightmost Derivation**: replace the rightmost variable at each step

- A right-setential form occurs in a leftmost derivation

What is an unambiguous grammar?

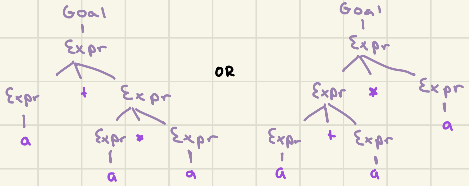
→ Only one possible parse tree (regardless of derivation order)

→ If derivation order is fixed, there should only be one possible derivation.

Example of an ambiguous vs unambiguous expression?

→ Ambigous grammar G_5 , and generating the expression a + a * a :

Goal → Expr
Expr → Expr + Expr
| Expr * Expr
| (Expr)
| a



CTD next page

Example of an ambiguous vs unambiguous expression?

→ Unambigous grammar G_6:

Goal → Expr

Expr → Expr + Term

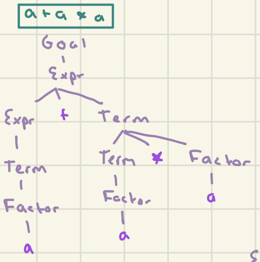
| Term

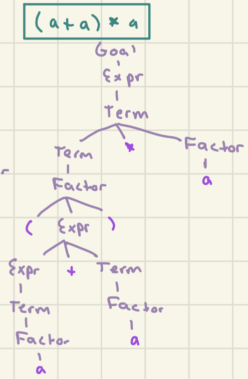
Term → Term * Factor

| Factor

Factor → (Expr)

| a





→ G_6 preserves PEMDAS/operation order unambiguously

What is semantic ambiguity?

→ When ambiguity in the grammar reflects true ambiguity in the language (e.g., overloading)

→ Ex: Fortran Grammar: Term → name

| Call

| ArrayRef

Call → name (ExprList)

ArrayRef → name (ExprList)

- Given a = f(17) ... we don't know whether this is a call to function f, or a reference to ellement 17 of array f.

How do we handle semantic ambiguity?

→ Solve outside the grammar:

- Use declaration info to return diff categories from the scanner
- Parse an imprecise grammar & fix it during type checking.

→ Change language to be unambigouous

- e.g., using different syntax for array refs, like [ ]

→ In Summary:

- handle language ambiguity through type information
- handle grammar ambiguity by rewriting the grammar.

Final notes on ambiguity?

→ The grammar for a programming lang. should be unambiguous.

→ Some ambiguous grammars can be rewritten to be ambigous & still capture the same language.

→ But some langs are inherently ambiguous. For ex, $\{a^i b^j c^k \mid i = j \text{ OR } j = k\}$

→ A CFG G is ambiguous if, for some sentence ℓ in L(G):

- it can produce more than one leftmost derivation for ℓ
- it can produce more than one rightmost derivation for ℓ
- it can produce more than one parse tree for ℓ.

→ In an unambiguous grammar, the leftmost & rightmost derivations may differ.

BUT, they must have the same parse tree.

- Precedence -

How do you add precedence to a grammar?

1. Decide how many levels of precedence are needed (e.g. PEMDAS)
2. Create a nonterminal (NT) for each level
3. Make higher precedence operators go through the productions for lower-precedence operators

→ Ex: ( ), +, -, *, and / :

- Level 1 = ( )
- Level 2 = *, /
- Level 3 = +, -

Example grammar implementing precedence

```
Goal → Expr
Expr → Expr + Term
     | Expr - Term
     | Term
Term → Term * Factor
     | Term ÷ Factor
     | Factor
Factor → ( Expr )
       | num
       | name
```

Final note on CFGs?

→ the language = the set of strings

→ The grammar :

- defines the set of strings
- encodes info about the structure

→ 2 grammars may yield the same language

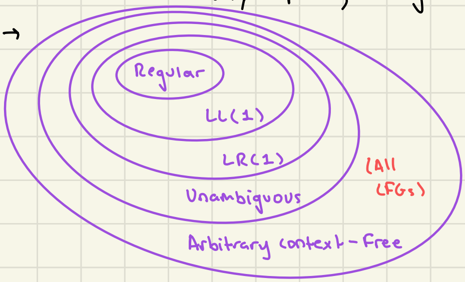
- with or w/o ambiguity
- with different heirarchical structure

- Classes of CFGs -

How can we classify CFGs?

→ based on the difficulty of parsing their grammars.

→



What are the classifications?

→ Regular (RG): CFGs that generate regular langs

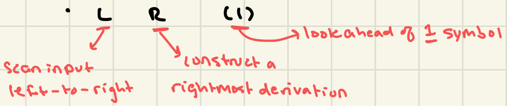
- equivalent to REs (encode langs recognizable by DFAs)
- Primary use in compiler construction = for specifying scanners

→ Arbitrary CFGs: require more time to parse than the more restricted LR(1) or LL(1) grammars (for ex, $O(n^3)$).

- A lang is context free if it can be recognized by an NPDA

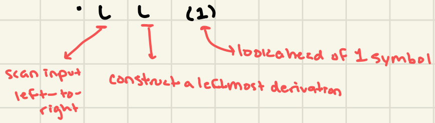
→ LR(1) grammars:

- able to be parsed bottom-up, in a linear L-to-R scan, looking at most one word ahead of the current input symbol
- A language is LR(k) if it can be recognized by a DPDA
- LR(k) and LR(1) grammars can express the same set of langs.
-



→ LL(1) grammars:

- able to be parsed top-down, in a linear L-to-R scan, looking at most one word ahead of the current input symbol
- can be parsed w/ either a hand-coded recursive-descent parser, or a generated one
- Many programming langs can be expressed in an LL(1) grammar.
-



→ Almost all programming lang. constructs can be expressed in LR(1), and often LL(1) form.

Top-Down Parsing

What does the pseudocode for a top-down parser look like?

→ RECALL: Top-down parsing = try to recreate sentence & using G.

(let NT = Nonterminal, aka symbol (not in Σ))

```
root = node for start symbol S
focus = root
stack.push(null) push null onto the stack
word = NextWord()
while (true):
    if (focus is NT):
        pick rule to expand focus, $A \rightarrow \beta_1 \beta_2 \beta_3 \dots \beta_n$
        build nodes for $\beta_1 \beta_2 \beta_3 \dots \beta_n$ as children of focus
        push $\beta_1 \beta_{n-1} \beta_{n-2} \dots \beta_2$ onto the stack
        focus = $\beta_1$
    else if (word matches focus):
        word = NextWord()
        focus = stack.pop() pop value off of stack
    else if (word == eof && focus == null):
        accept input
        return root
    else:
        backtrack (set focus = parent node & try
    a diff production. If no untried ones remain, return error
```

What is the "lower fringe"?

→ In a tree: it ≈ a sentential form in the table.

What is the focus?

→ The node in a tree that is working to expand

What is the lookahead symbol?

→ The next word in the string to match.

What are the requirements for a grammar to be Top-Down parsable?

→ Not left recursive: left recursion leads to infinite expansions

- transform grammar to be right recursive

→ Backtrack-free (aka predictive OR LL(1))

- backtracking is expensive
- transform grammar to be predictive
- not all languages have an LL(1) grammar

What is the focus symbol?

→ in a top-down parser, you keep a stack of symbols you still need to match/expand.

→ Focus symbol: the symbol at the point you're about to act on.

- If its terminal: try to match it with the next input token (the lookahead)
- If its nonterminal: expand it using one of its productions

- Left Recursion -

What is a left-recursive grammar?

→ a rule in a CFG is left-recursive if the 1st symbol on its right-hand side is the symbol on its left-hand side, or can derive that symbol.

→ Ex:

$\text{Fee} \rightarrow \text{Fee}\ a$
$\quad\quad\ \ |\ \beta$

• Fee is on left-hand side

Direct Left Recursion

$a \rightarrow \beta$
$\beta \rightarrow \gamma$
$\gamma \rightarrow a S$

• β can be used to derive a

Indirect Left Recursion

→ A CFG is left-recursive if,

for some $A \in NT$ and $B \in (T \cup NT)^*$, $A \rightarrow^+ AB$] formal defn

How do we eliminate left-recursion?

What are the steps?

→ Transform grammar from left to right-recursive :

→ Naive rewriting changes the associativity

1) Forward substitution: change indirect LR → direct LR

2) Transformation: change direct LR → RR ...assumes that the grammar has no cycles & no ε-productions

Example?

→ classic expression grammar G_1 :

Goal → Expr
Expr → Expr + Term
| Expr - Term
| Term
Term → Term * Factor
| Term ÷ Factor
| Factor
Factor → (Expr)
| num
| name

left-recursive rule: 1st symbol on right (Expr) = symbol on left

→ Rewritten R-recursive rules for G_1:

Goal → Expr
Expr → Term Expr'
Expr' → + Term Expr'
| - Term Expr'
| ε
Term → Factor Term'
Term' → x Factor Term'
| ÷ Factor Term'
| ε

→ by moving recursion to right-side, infinite loop is avoided:

	Rule	Transformed
Expr	2	Term Expr'
	3	Term + Term Expr'
	4	Term + Term - Term Expr'
	5	Term + Term - Term

→ Why? Because left-recursive productions don't generate a leading terminal symbol.

What do we have to watch out for when making a grammar right-recursive?

→ Preserving Associativity: eliminating left recursion must maintain associativity.

→ Ex : Converting left-recursive G_1: Expr → Expr - name | name into G_2: Expr → name - Expr | name would be a naive attempt.

- Predictive Parsing -

What is backtracking?

→ When the parser expands the lower fringe with the wrong production, eventually encounters a mismatch between that fringe & the parse tree's leaves (aka the terminals/string of the sentence itself), and then has to undo all of the actions that built the wrong fringe. And try other productions.

→ Backtracking - expanding, retracting, & re-expanding the fringe - wastes time & effort.

How do we make a parser backtrack-free?

→ A top-down parser may choose the wrong production & need to backtrack

→ Better: look ahead in the input to pick the right production

→ In general, an arbitrarily large amount of look-ahead may be needed

- Look-ahead handled by Cocke-Younger, Kasumi, or Earley's algorithms

What is the effect of using lookahead?

→ Large subclasses of CFGs can be parsed with limited lookahead

- e.g., LL(1) & LR(1) grammars. Most prog. lang constructs fall in these subclasses

→ When parser goes to select next rule (in trying to recreate sentence), it can consider both the focus symbol & the next input (lookahead) symbol.

→ w/ one-symbol lookahead, parser can disambiguate all of the choices that arise in parsing the right-recursive expression grammar.

What is predictive parsing?

→ Given $A \rightarrow \alpha \mid \beta$ and the next token in the stream, the parser can correctly choose between α and β

→ Additional definitions needed: FIRST sets, FOLLOW sets

→ Backtrack-free grammar = Predictive grammar.

What is a FIRST set?

→ Let α = some RHS symbol in grammar G.

RECALL: T = terminals = Σ (alphabet)

→ FIRST(α) = the set of tokens that appear as the first symbol in some string that derives from α:

- FIRST : $T \cup NT \cup \{\varepsilon, eof\} \rightarrow$ PowerSet $(T \cup \{\varepsilon, eof\})$
- $x \in$ FIRST(α) iff $\alpha \Rightarrow^* x\gamma$, for some γ

→ For every terminal β, FIRST(β) = $\{\beta\}$. e.g., FIRST(num) = {num}

Example of FIRST sets?

→ Consider grammar G_1:

```
Goal   → Expr

Expr   → Term Expr'

Expr'  → + Term Expr'
       | - Term Expr'
       | ε

Term   → Factor Term'

Term'  → x Factor Term'
       | ÷ Factor Term'
       | ε

Factor → ( Expr )
       | num
       | name
```

1. Compute FIRST sets for terminal symbols

   FIRST(Expr) = { (, name, num }

   - Expr goes to Term^{1st} Expr'. Term goes to Factor^{1st} Term'. Factor goes to (Expr), num, name

   FIRST(Expr') = { +, -, ε }

   FIRST(Term) = { (, num, name }

2. For each production $A \rightarrow \beta$, compute FIRST(β)
3. Iterate until a fixed point is reached.

Example to understand intuition behind predictive (LL(1)) parsing with FIRST sets?

(Inputs & lookaheads listed at each step are the ones at the END of step)

→ take this grammar: $S \to c \mid aSb$

→ Say the parser has string aacbb that it is trying to "recreate"

0. START

Stack (topmost = last in)

$
S

Input = a a c b b $

lookahead = a

→ Build the LL(1) FIRST table: For each production $A \to \alpha$, compute FIRST(α):

α	Rule No.	FIRST(α)
c	1	{ c }
aSb	2	{ a }

1. → Focus symbol = top of stack = Stack.pop() = S ... nonterminal.

→ Find the production $S \to x$ s.t. $a \in$ FIRST(x) : Parser checks lookup table, and now knows that it should pursue Rule 2, NOT Rule 1.

→ Pop S ; push RHS aSb in reverse so that leftmost a ends up on top:

Stack.push(b); stack.push(S); stack.push(a);

Stack

$
a
S
b

Input: a a c b b $

lookahead = a

2. → Focus symbol = Stack.pop() = a ... terminal.

→ Since focus symbol is terminal, match it with lookahead a

→ Pop a ; advance input

Stack

$
S
b

Input: ~~a~~ a c b b $

lookahead = a

3. → Focus symbol = S ... nonterminal, Lookahead = a so choose Rule 2.

→ Pop S, push aSb in reverse (same as step 1)

Stack

$
a
S
b
b

Input: a c b b $

lookahead = a

4. → Focus symbol = a ... Same as step 2. Pop a b/c matches lookahead.

Stack

$
S
b
b

Input = ~~a~~ c b b $

lookahead = c

5. → Focus symbol = S. Parser checks lookup table & chooses Rule 1! $S \to c$

→ Pop S and push RHS c to stack

Stack

$
c
b
b

Input = c b b $

lookahead = c

6. Focus symbol = c (terminal). Match w/ lookahead. Pop c & advance input.

Stack
$
b
b

Input: b b $

lookahead = b

7. Focus symbol = b (terminal). Match w/ lookahead. Pop b & advance input.

Stack
$
b

Input: ~~b~~ b $

lookahead = b

8. Focus symbol = b (terminal). Match w/ lookahead. Pop b & advance input.

Stack
$

Input: ~~b~~ $

lookahead = ∅

→ All done! With no backtracking

What is the predictive/LL(1) property?

→ If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, then $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$

- Allows parser to make correct choice with single look-ahead token
- Does not account for ε-productions

What are ε-productions?

→ Rules that expand to ε, e.g. $\alpha \rightarrow \varepsilon$.

if $A \rightarrow \alpha | B$ and $\varepsilon \in \text{FIRST}(\alpha)$, how should the parser choose which rule to apply?

→ Ex:

$A \rightarrow \alpha | \beta$
$\alpha \rightarrow cd | \varepsilon$

$\text{FIRST}(\alpha) = \{c, \varepsilon\}$

$\text{FIRST}(A) = \{c, \varepsilon\}$

→ Apply ε when the look-ahead symbol appears in no other production's FIRST sets

→ FOLLOW sets are needed.

What is a FOLLOW set?

→ FOLLOW : NT → PowerSet(T ∪ {eof})

→ For a nonterminal α, FOLLOW(α) is the set of terminal symbols that can appear immediately after α in some sentential form.

→ FOLLOW(α) = set of all words that can occur to the immediate right of a string derived from α

→ Purpose: parser needs to know which words can appear as the leading symbol after a valid application of a rule.

How does the parser compute FOLLOW sets?

1. For the topmost rule (ex: Goal → Expr) : Set FOLLOW(Goal) = {eoF}
2. For each NT, if $\beta_i \beta_{i+1}$ appears in the RHS, then FOLLOW(B_i) contains FIRST(β_{i+1})
3. Iterate until a fixed point is reached.

→ Parser utilizes pre-calculated lookup table of all FIRST sets.

What is the pseudocode for computing FOLLOW sets?

```
for each $A \in NT$:
    $FOLLOW(A) = \emptyset$
$FOLLOW(S) = \{eof\}$        $S$ = start symbol
while (FOLLOW sets are still changing):
    for each $p \in P$ of the form $A \rightarrow \beta_1 \beta_2 \dots \beta_k$:
        TRAILER = FOLLOW(A)
        for $i = k, k-1, \dots 1$: (aka from right-to-left of the RHS of a production)
            if $\beta_i \in NT$:
                $FOLLOW(\beta_i) = FOLLOW(\beta_i) \cup TRAILER$
                if $\varepsilon \in FIRST(\beta_i)$:
                    $TRAILER = TRAILER \cup (FIRST(\beta_i) - \varepsilon)$
                else: $TRAILER = FIRST(\beta_i)$
            else:
                $TRAILER = FIRST(\beta_i)$ (aka $\{\beta_i\}$ since $FIRST(x) = \{x\}$
                                                for some terminal $x$)
```

Example?

→ Classic expression grammar:

$Goal \rightarrow Expr$

$Expr \rightarrow Term\ Expr'$

$Expr' \rightarrow +\ Term\ Expr'$
$\quad\quad | \ -\ Term\ Expr'$
$\quad\quad | \ \varepsilon$

$Term \rightarrow Factor\ Term'$

$Term' \rightarrow \times\ Factor\ Term'$
$\quad\quad | \ \div\ Factor\ Term'$
$\quad\quad | \ \varepsilon$

$Factor \rightarrow (\ Expr\ )$
$\quad\quad | \ num$
$\quad\quad | \ name$

NT	FIRST (NT)	FOLLOW (NT)
Goal	(, num, name	eof
Expr	(, num, name	eof,)
Expr'	+, −, ε	eof,)
Term	(, num, name	eof,), +, −
Term'	×, ÷, ε	eof,), +, −
Factor	(, num, name	eof,), +, −, ×, ÷

What is the (full) Predictive, or LL(1) Property?

→ let START $(A \rightarrow \alpha)$ be defined as:

- $FIRST(\alpha) \cup FOLLOW(A)$, if $\varepsilon \in FIRST(\alpha)$, or
- $FIRST(\alpha)$ otherwise.

→ Given a grammar G and its FIRST and FOLLOW sets, we say that G has the LL(1) property i.f.f. $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

$START(A \rightarrow \alpha) \cap START(A \rightarrow \beta) = \emptyset$

→ A grammar with the LL(1) property is a predictively parsable grammar.

Example of a Non-LL(1) Grammar?

Goal → S

S → A | B

A → (A) | a

B → (B) | b

NT	FIRST(NT)
S	(, a, b
A	(, a
B	(, b

→ let $A \to \alpha \approx S \to A$ and $A \to \beta \approx S \to B$

→ START(S → A) = FIRST(A) = {(, a}

→ START(S → B) = FIRST(B) = {(, b}

→ START(S → A) ∩ START(S → B) = (. Since it isn't ∅, this grammar doesn't satisfy LL(1) property!

What is Left Factoring?

→ Transforming a non-LL(1) grammar into an LL(1) grammar

→ Not all grammars can be transformed

How does it work?

→ Assume a grammar G with productions $A \to \alpha\beta_1$ and $A \to \alpha\beta_2$

→ If α derives anything other than ε, G is not LL(1) b/c doesn't satisfy property.

1. Left-Factor to pull the common prefix, α, into a separate production:

   $A \to \alpha A'$, $A' \to \beta_1$, and $A' \to \beta_2$

2. If $\text{START}(A' \to \beta_1) \cap \text{START}(A' \to \beta_2) = \emptyset$, G MAY be LL(1)

Example of Left Factoring?

→ Non-LL(1) Grammar G:

Factor → name | name [ArgList] | name (ArgList)

ArgList → Expr MoreArgs

MoreArgs → , Expr MoreArgs | ε

→ Left-Factored:

Factor → name Args

Args → [ArgList] | (ArgList) | ε

ArgList → Expr MoreArgs

MoreArgs → , Expr MoreArgs | ε

How does automatic parser generation work?

→ Write a CFG and generate an excellent parser

→ The parser generator analyzes the CFG & encodes the information into:

1) Tables to drive a skeleton parser ; OR

2) A direct-code recursive descent parser

→ Generated parsers are efficient (asymptotic) & fast (constants)

→ Parser generators handle most prog. lang. features

- LR(1) is more general
- LL(1) accepts a strictly smaller class of langs & grammars

→ Standard work-arounds cover the features that do not immediately work

RECAP: What is Backtrack-Free Top-Down Parsing?

→ Requires backtrack free (aka predictive) grammar

→ Build FIRST and FOLLOW sets

→ Use the LL(1) condition/property to know whether a grammar is backtrack-free

→ Use left-factoring to rewrite a grammar to make it backtrack free

Top-Down Parsing

→ Recursive Descent

→ Table Driven

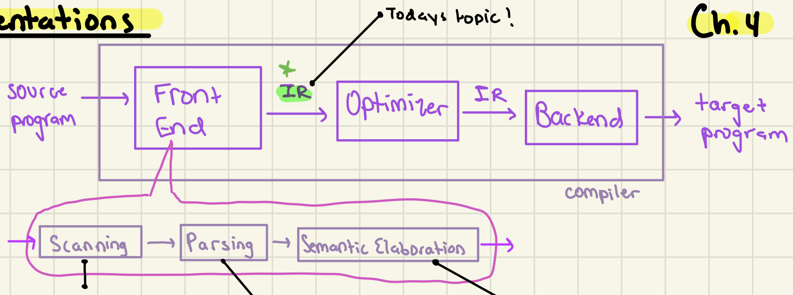
→

(SKIPPED - see end of bottom-up lecture. AND top-down lecture)

Intermediate Representations

Ch. 4

RECAP: Where are we now, in the compiler?



Scanning:
→ stream of chars → stream of words
→ outputs <category, lexeme> pairs
→ COVERED ✓

Parsing:
→ is stream of words a sentence in the source language?
→ maps stream of words → sentences in source lang
→ COVERED ✓

Semantic Elaboration:
→ static semantic analysis
→ is the sentence (statically) meaningful?
→ Output: I.R.
→ NOT YET COVERED

What is the IR?

→ The authoritative version of the program

Front End	Optimizer	Backend
↓	↓	↓
Produces IR	traverses & transforms IR	transforms IR into native code

→ Encompasses program representation, and derived knowledgo:

* symbol tables * storage maps

→ The compiler can manipulate only those details that are represented in the IR

What is the taxonomy (classification) of an IR?

→ Structural Organization: graphical, linear, hybrid

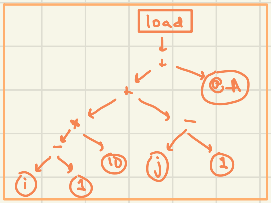
→ Level of Abstraction: near-source form, near-machine code form

→ Mode of Use: definitive, derivative

What are the ways to structurally organize an IR?

→ Graphical IR (e.g. parse trees, ASTs) ↳ aka MSO

- encodes program structure
- useful in optimization & in source-to-source translators
- Graphs can be large if not careful



good for instruction selection

→ Linear IRs (e.g. 3-address code, stack machine code)

- simple, compact data structures
- easy to rearrange
- no structural information

```
load2DArray A,i,j
```

good for code relocation

```
loadI 1        => r1
sub   rj, r1   => r2
loadI 10       => r3
mult  r2, r3   => r4
sub   ri, r1   => r5
add   r4, r5   => r6
loadI @A       => r7
add   r7, r4   => r8
load  r8       => rAij
```

good for address optimization

→ Hybrid IRs (e.g. control-flow graphs)

- combine graphs & linear code

What are the modes of use?

→ Definitive IR: primary representation of the code, transmitted from one pass to the next

→ Derivative IR: built for a specific temporary purpose. Used within a single pass of the compiler.

What are the key properties of an IR?

→ Ease of generation

→ Ease of manipulation

→ cost of manipulation

→ procedure size

→ IR design affects the speed & efficiency of a computer

– Graphical IRs –

How do Parse Trees weigh up, as IRs?

→ RECALL: Parse tree: concrete rep. of the derivation

→ Captures the structure of the grammar

Pros?

→ Includes syntactic details of the input program

→ Useful in source-to-source compilers

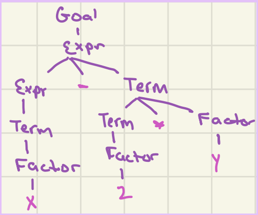
Cons?

→ Tend to be inefficient

→ Explicitly states what could be represented implicitly

Example?

→ Ex:



← x – 2 * y

Goal → Expr
Expr → Term Expr'
Expr' → + Term Expr'
| – Term Expr'
| ε
Term → Factor Term'
Term' → x Factor Term'
| ÷ Factor Term'
| ε
Factor → (Expr)
| num
| name

How are ASTs, as IRs?

→ AST = a parse tree minus most nodes for non-terminal symbols (many fewer nodes than a syntax tree)

Pros?

→ Could regenerate source code in a treewalk.

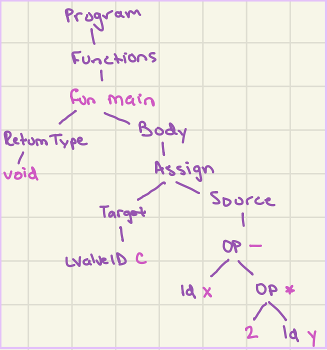
→ Often used as initial IR

Cons?

→ AST node sizes tend to grow large

Example?

→ For the expression c = x – 2 * y; →



How do we use Directed Acyclic Graphs (DAGs) as IRs?

→ DAG IR = AST with a unique node for each value.

- e.g., AST with sharing

Pros?

→ Given 2 instances of the same expr, the compiler may be able to arrange the code to evaluate it only once. (For ex, only performing evaluation of a * 2 once but using the result twice, in an expression a * 2 + a * 2 * b)

→ Encodes redundancy. e.g., with this example, the AST would contain 2 distinct copies of the expression a * 2.

- DAG avoids this duplication by reusing identical subtrees & allowing nodes to have multiple parents.

→ Makes sharing explicit

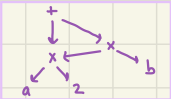
Cons?

→ Compiler must <u>prove</u> an expressions value doesn't change (in order to reuse)

- easy if there are assignments or function calls

Example?

→ a * 2 + a * 2 * b →



How do the trees get implemented?

→ Nodes of various types & arities, connected by edges

→ Allocating nodes of diff sizes complicates allocation & fragmentation in the heap (malloc())

→ A compiler executes just often enough to justify a careful implementation.

What is a dependence graph?

→ Encodes the flow of values in a block

→ Nodes = operations

→ Edges represent a flow of values

→ Often built as a secondary IR for scheduling or optimization

→ Used as a primary IR in some JITs (just-in-time compilation)

What are call graphs?

→ Represent the flow of control between procedures

→ Nodes = procedures

→ Edges represent individual calls (each edge represents a different calling environment)

→ Used to optimize multiple procedures together

→ Complex call patterns create complex graphs (e.g., recursion can create cycles)

→ Some interprocedural optimizations traverse & change the call graph (e.g., inline substitution)

-Linear IRs-

What is 3 address code?

→ Statements of form $x \leftarrow y \text{ op } z$

- 1 operator (op)
- at most 3 names (x, y, & z)

→ Used in many modern compilers

→ Resembles many real machines

→ Introduces a new set of names:

e.g. $z \leftarrow x - 2 * y$ becomes $t \leftarrow 2 * y$, $z \leftarrow x - t$

How do we implement 3-address codes?

→ As a set of quadruples: A table of $k * 4$ small integers

→ Each quadruple is represented w/ 4 fields: operator, 2 operands (or sources), and a destination

→ Opcodes & operands stored as small integers

Pros of quadruples?

→ Simple record structure → Easy to reorder

→ Explicit names

```
t1 ← 2
t2 ← b
t3 ← t1 × t2
t4 ← a
t5 ← t4 - t3
```

3-address code for a - 2 × b

Example?

→ For a - 2 × b:

```
load  r1  b
loadi r2  2
mult  r3  r2  r1
load  r4  a
sub   r5  r4  r3
```

equivalent RISC assembly

Opcode	Op1	Op2	Op3
load	1	b	
loadi	2	2	
mult	3	2	1
load	4	a	
sub	5	4	3

Quadruples

Hybrid IRs

What are control flow graphs?

→ Models the transfer of control in the procedure

→ Nodes = basic blocks

→ Edges = transfers of control

- branches, jumps, predicated instructions

How do we use them?

→ Use CF Graph (CFG) to represent control flow, & a linear IR (like 3-addr code) to represent code in blocks

→ Graph benefit: easy navigation between blocks

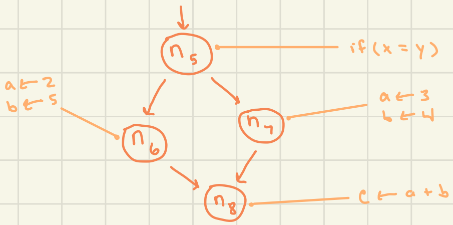
→ Linear IR benefit: explicit, low-level detail & operation sequence

→ Useful for program analysis

Example?

```
if (x = y)
    a ← 2
    b ← 5
else
    a ← 3
    b ← 4
c ← a + b
```

⟶



How do CFGs compare to ASTs?

→ ASTs show grammatical structure, while CFGs show control flow

→ Eg:

```
while i < 100
        stmt 1
stmt 2
```

— Name Spaces & Scopes —

What are namespaces & scopes?

→ A fundamental abstraction in most programming languages, that allow us to build real systems

- E.g., use libraries w/o concern that a name was already used

→ New scopes create a clean slate for naming

How does the IR track them?

→ IR includes a collection of annotations; it records names, values, constants, & locations for all program entities

How are the IRs records used?

→ Mapping of an instance of a name ⟶ a specific declaration

→ Searching hierarchies to model scope rules & inheritance

What are some key capabilities?

→ Compile-time map from a name to a specific entity (declaration, definition)

- scopes, visibility
- binding & general organization

→ Runtime mechanism to find & access a given entity

- storage assignment
- data-area addressability

How does the compiler map a name to an entity (at compile-time)?

→ Compiler needs info to generate an access (load, store, or call)

→ To translate a reference to an entity, the compiler needs to know about the entity

→ SOLN: Derive info in the compiler's front-end, & store that info in some handy form for later use.

- This is what the IR is for! IR is where we "store that info"

What are symbol tables?

→ Compile-time data structure for storing info for mapping names to entities

→ Maps names (symbols) to their attributes

- This is how the compiler resolves names

1. Initial info is given where the entity is defined. AKA:
   - A variable's declaration or first use
   - A procedore's definition or its prototype
   - An object's class declaration (plus its superclass, super-superclass...)
2. Compiler constructs a type signature to represent that entity
   - It may also assign storage to the entity

How are symbol tables (STs) structured?

→ entry exists for every symbol, & contains associated attributes:

- lexeme
- type
- address
- length
- storage class (local, static, global, constant, etc)

conceptual example of an ST?

Name	Type	Addr	Len	Storage Class
y	int	8	4	local
w	float	@_w	4	static
x	char	12	0	static
z	double	0	8	local

Naming Environment

What is scope?

→ DEFN: The region of a program where a given name can be accessed

→ A contiguous set of statements in which a name is declared, visible, & accessible

What are the 2 main types?

→ Lexical Scope → Inheritance heirarchies

What is name resolution?

→ mapping each var reference to its specific declaration

→ Compiler does this using either lexically-scoped STs, or Inheritance hierarchy STs

— Lexical Scopes —

What are Lexical Scopes?

→ DEFN: Scopes that nest in the order in which they appear in the source code

→ Main rule:

An occurrence of name n in the prog. at point p, refers to the entity named n that was created — implicitly or explicitly — in the scope that is lexically closest to p.

How do scopes present in programs?

→ Nested regions of code — e.g. procedures, blocks, structs, modules

→ Each scope creates a new name space

→ Scoping is tied to program semantics — languages may differ in scoping rules

- e.g. rules for nesting, inheriting, or obscuring names from outer scopes

What are Algol-like languages?

→ ALGOL = Algorithmic language (name comes from the family of languages developed in 1958)

→ Examples: Algol, Pascal, Fortran, C, Scheme, Java, C++

How do Algol-like languages handle scoping?

→ Each scope creates its own namespace (e.g. procedure, block, structure)

- Procedure scopes typically contain formal parameters

→ Almost any name can be declared locally

→ Local names obscure identical non-local names

- e.g. in this code:

```
int i = 10;
for(int i=0 ;i<3;i++){
      ...}
```

everything inside the scope of the for-loop only recognizes the variable named "i" that was declared inside of it (in the guard line)

→ Local names cannot be seen outside their scope

→ Scopes are searched in nesting order, inner scope to outer scope

How do common Algol-like langs behave?

→ Algol, Pascal: nested procedures, often with deep nesting

→ Fortran: • local, static, & named global scopes

- no nested procedures

→ C: added a little to Fortran

→ Scheme: global, procedure-wide, & nested scopes

How does Python handle lexical scopes?

→ Python has 3 nested lexical scopes : • builtin • global (module)

• local (function); functions can be nested

→ Declaration before use not required – First use is defining use

→ If defining use is :

- Assignment : creates new local symbol with inferred type
- Reference : binds to or creates a global symbol
- Nonlocal declaration : ensures name is in the global scope

How does the compiler model lexical scopes?

→ Compiler builds model of the scope structure of the code

→ The parser builds a set of tables & search paths

- one table for each scope
- names declared in a scope go into the scope's table
- search paths link the tables together

→ The parser builds a distinct path as it enters or leaves a scope

- Path : instantiates the namespace according to scope rules
- Preserve the search paths with the code

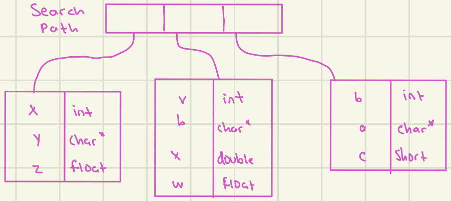
→ Compiler preserves the tables for later use (debugging)

How do scoped symbol tables work?

→ New table for each scope ; Individual tables are hash tables with O(1) lookups

→ Chain tables together into a "search path"

- Search starts with 1st table, & "fails" its way up the path until it finds the name, or falls off the end of the path.



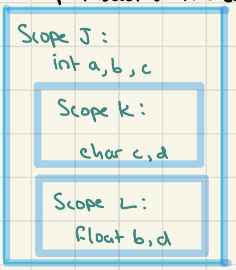
x	int
y	char*
z	float

v	int
b	char*
x	double
w	float

b	int
a	char*
c	short

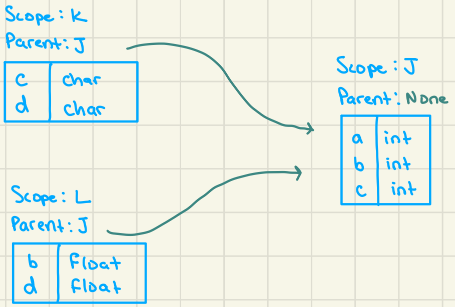
Example of scoped STs?

→ "Scope Model" of the code :



```
Scope J :
   int a,b,c
   Scope K :
      char c,d
   Scope L :
      float b,d
```

→ Tables to model the name space :



Scope : K
Parent : J

c	char
d	char

Scope : L
Parent : J

b	float
d	float

Scope : J
Parent : None

a	int
b	int
c	int

How are the scoped tables built?

- → Source language defines the syntactic constructs that demarcate (set the limits/boundaries of) scope
  - e.g. new procedure, block, class decl, struct decl
- → Demarcated scope entry & exit:
  - Table is created & added to front of search path UPON block entry
  - Table is finalized on block exit
- → Variable Declaration:
  - An entry for the declared variable, along with its attributes, is created in the current table
  - langs that don't require type declarations infer type info (e.g. Python)
- → Variable Reference: a lookup upon the current search path (e.g. from current scope, moving outwards) is triggered

– Inheritance Hierarchies –

What are inheritance hierarchies? (IH)

- → Another type of naming environment
- → A data-centric naming scheme: data & code is accessed relative to the **object they belong to**, rather than relative to the **current procedure**

When are they used?

- → In object-oriented languages (eg C++, Java, Python)
- → subclasses & superclasses define the inheritance hierarchy, which is orthogonal to the lexical hierarchy

RECALL: what are OO-langs?

- → Object: an abstraction with members (data, code, other objects); i.e. a class instance. maintains internal state
- → Class: A collection of objects w/ the same structure. Defines data & code members of each instance
- → Inheritance: A relationship defining a partial order on the namespaces of classes
- → Receiver: methods are invoked relative to an object -- the method's receiver
  - e.g. "self." for Python, or "this." for Java

How does inheritance work in OO-langs?

→ For ex:

```
Class Alpha {...}
Class Beta extends Alpha {...}
```

- → Beta objects inherit code, data members from Alpha definition
- → Beta may redefine names from Alpha
- → Methods in Alpha must work correctly on Beta objects (provided the method is visible in Beta)
- → Beta may itself be extended – this creates a tree of inheritance
- → Beta may inherit from multiple immediate superclasses – this creates an acyclic graph of inheritance

→ Given a named member, compiler searches the inheritance hierarchy to resolve the name.

How does hierarchical scoping work?

→ Each class definition creates a new scope

→ Data members, code members are in the scope

- Visibility parameters can affect scope

→ Scope tables & search paths work as in lexical scoping

Example?

```
Class Point {
    public int x;
    private int y;
    public void draw () {...}
    public void move() {...}
Class ColorPoint extends Point {
    private Color c;
    public void draw (){...}
    public void setc (Color x) {
        this.c = x };
}
```

Scope Tables

Class: Class
Superclass: None

Class: Point
Superclass: Class

x	int	public
y	int	private
draw	void()	public
Move	void()	public

Class: ColorPoint
Superclass: Point

c	Color	private
draw	void()	public
setc	void()	public

What is compile-time name resolution?

→ At compile time, the compiler looks at STs and etc. & maps each name to its address in memory.

→ Runtime name resolution: The lookups & etc. are done during execution.

What is a closed class structure?

→ An OO language has a closed-class structure if it requires that class definitions/ structures must be present & fixed at compile-time.

→ OOLs with closed class structure are able to resolve names at compile-time

Characteristics?

→ Hierarchy known at compile time

→ Compiler can build models & emit code for all references

EX: C++

→ Virtual functions force runtime binding

What is an open class structure?

→ Characteristic of OOLs that allow the running program to change its class structure at runtime (for ex, importing classes in Java)

→ OOLs with open class structure must resolve names at runtime.

EX: Java

Characteristics?

→ Hierarchy can change at runtime

→ Compiler can build models & emit code for some references.

→ Compiler must emit code to resolve some references at runtime

How does dynamic (runtime) name resolution work?

→ Runtime creates & maintains the namespace tables

- higher cost for references
- New tables & search paths created as hierarchy changes

→ Frequency of change is critical

(unfinished notes from slide)

RECAP: Lexical vs Hierarchical Scoping?

→ Lexical: Given an unqualified name n in procedure p:

- compiler first searches lexical scope for n, then searches hierarchical scope

→ Hierarchical: Given a qualified name my_obj.n

- compiler first resolves the name my_obj (aka an object) lexically; eg looking for where it was instantiated
- Compiler then resolves the name n using my_obj's hierarchical scope

– Visibility –

What is visibility?

→ A name is **visible** at point p in the src code if it can be referenced at point p.

→ Visibility is orthogonal to lexical & heirarchical scoping

→ Lang. defines programmer's level of control over visibility

→ Visibility info included in ST entries

EX: How is visibility defined in C?

→ C: Static variables: Lifetime = the entire execution

- Inside a procedure, value is preserved across invocations

→ Visibility restricted to current & nested scopes

- Outside a proc: visibility restricted to current file

Ex: How is visibility defined in Java?

→ Java: public, private, protected, & default methods or data members

- public: visible anywhere in the prog
- private: visible only in the enclosing class
- protected: visible in enclosing class, other classes in same package, & any subclasses in any package
- default: visible in enclosing class, other classes in same package

→ Ex: Given MyClass, member MyMethod(), and instance Obj, Obj.myMethod() can access:

- local vars declared in myMethod()
- data members of both obj & MyClass
- public & protected vars of any of MyClass' superclasses
- classes defined in same pkg as MyClass, or in any explicitly imported package
- public class & instance vars of imported classes
- package class & instance vars in the pkg containing MyClass

Type Information

Ch.5

What are types?

- → An abstract category defined by a set of properties. All members of the category have the same set of properties.

What is a type system?

- → The set of types, their properties, & the lang's semantic definitions that use types
- → Provides additional context & rules of behavior over the CFG of the lang, which are useful to the compiler, e.g.
  - checking that operator & operands are conformable (eg MS1!)
  - checking function sigs
  - garbage collection

What does a type system consist of?

- → Base Types
- → Rules for constructing types
- → Tests for type equivalence
- → Rules for type inference

What are the Base Types?

- → Common: numbers, characters, booleans
  - processor support exists
  - various varieties
- → Additional (e.g., not universal; language dependent):
  - pointers, recursive lists, rationals, strings, maps

What are constructed types?

- → Aggregate types that allow the programmer to organize & use base data as higher-level constructs.
- → E.g., arrays, strings, enums, structs, unions, records

What is an array (high level)?

- → Groups multiple entities of the same type. Indices give each entity a unique name (eg `A[2][3]` is unique).

What are strings (high level)?

- → More than just an array of characters
- → Concatenation (eg `"hello" + "world"`); Comparison (e.g. `"Fee" < "Fie"`)
- → Length of string vs length of allocated memory

What are enums (high level)?

- → A set of self-documenting names for a small set of constants, e.g.

```
enum Day {Mon, Tues, Wed, ...}
```

- → Compiler maps distinct elements to distinct values.

What are structures?

- → Group multiple entities of arbitrary type
- → Elements are given explicit names, e.g. "`int pid;`"
- → Type is the ordered product of the element types.

Ex:

```
struct Student {
    int pid;
    char [] name; }
```

What are unions?

- → The disjunction of multiple entities
- → Type is disjunction of the type of component entities

Ex:

```
union Node {
    struct N1 one;
    struct N2 two; };
```

How is type equivalence tested?

→ Straightforward for base types

→ Language-dependent for constructed types

- name equivalence
- structure equivalence

How is type inference conducted?

→ Simple components (num, name): declare before use; infer by context (may require second pass of the IR)

→ Simple expressions: operator & operand type determine resulting type

→ Expressions involving function calls:

- requires type sig. of function
- function type may be parameterized

Example of parametric polymorphism?

→ filter: (a → bool) x list of a → list of a

filter (lambda x: x<5, [1, 4, 6, 3, 8]) → [6, 8]

Syntax-Driven Translation

Ch.5

RECAP: What does the IR do?

→ Building the IR encodes information about named entities that is needed by subsequent phases of the compiler :

- Namespace
- Storage
- Type

→ RECALL: Definitive IRs:

- AST
- 3-address code

What is syntax-driven translation? (S-DT)

→ A collection of computations tied to the grammatical structure of the source code

→ GOAL: Translate the source code into an IR

→ Computations may include:

- building the AST
- building the symbol tables & type checking
- emmitting 3-address code
- evaluating expressions

What are the 2 strategies for S-DT?

→ Build computation into initial parse – e.g., action rules in the grammar.

- the grammar's structure determines where the computation happens. e.g., taking the code where you are parsing, and adding in code that also performs relevant computations for the variables being parsed
- tying semantic actions to grammar productions

→ Walk the parse tree to compute

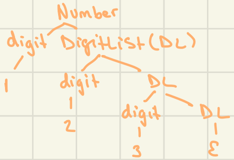
– Using Action Rules in the Grammar –

What is our "toy example" grammar?

→ Grammar for positive integers, G_2 (terminals = {digit}, NTs = {Number, DigitList}):

```
Number → digit DigitList
DigitList → digit DigitList | ε
```

• Ex: "123" ⟶



What might its parser look like?

```
boolean Number() {
    if (word == digit):
        word = NextWord()
        return DigitList()
    else: return false }
boolean DigitList() {
    if (word ==digit):
        word = NextWord()
        return DigitList()
  else: return true
```

How would we use SDT here?

→ SDT : The grammar determines the computation

→ Use points in the grammar to convert stream of lexemes into corresponding integer value. E.g., turning {<1, num>, <2, num>, <5, num>} into an actual integer with value 125.

How would we modify our parser for G_2 to perform translations?

→ Same control-flow as before, but now we are passing/carrying an attribute (value) through the calls in order to simultaneously accumulate/compute the integer value

```
(boolean,int) Number(value) {
    if (word == digit):
        value = CtoI(word)
        word = NextWord()
        return DigitList(value)
    else: return (false, invalid_value)}

(boolean,int) DigitList(value) {
    if (word == digit):
        value = (value * 10) + CtoI(word)
        word = NextWord()
        return DigitList(value)
    else: return (true, value)
```

CtoI = function that returns int value of a single char

What is another way to use action rules to build computation into initial parse?

→ Embed computations in the grammar!

→ Consider a left-recursive grammar G_3 for 1 or more digits:

```
Number → DigitList
DigitList → DigitList digit | digit
```

Example?

→ For each production, define semantic actions (eg "blocks"/code snippets). When the parser reduces a rule - for Ex recognizing DigitList digit & reducing it to DigitList - it then runs that block, using the already-computed value of the left DigitList, and the current digit, to compute the new DigitList value.

→ Implementation: let $$ = the NT on the LHS, and
$i = the ith symbol on the RHS

```
Number → DigitList              {return $1;}
DigitList → DigitList digit     {$$ = $1 * 10 + CtoI($2);}
          | digit               {$$ = CtoI($1);}
```

- Walking the Parse Tree -

What is the other strategy for SDT?

→ Creating PT first, then separately traversing it & performing computations.

→ e.g., for grammar G_3: Value(root).

```
    if (root is Number): return Value(root.child)
    elif (root is DigitList):
        return 10 * Value(root.left) + Value(root.right)
    elif (root is digit): return CtoI(root.lexeme)
```

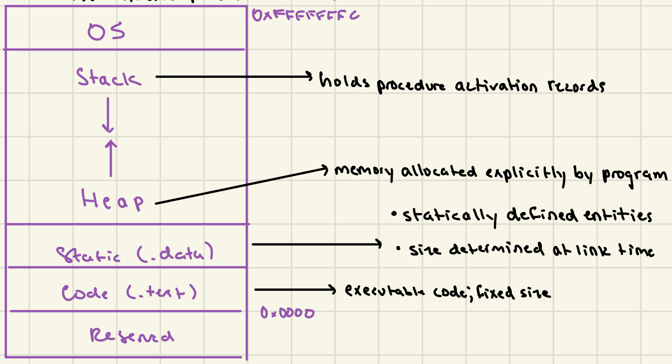
Storage Layout

Ch.5

RECAP: What is physical & virtual memory?

- → Physical addr. space: 2^{64} linearly addressable bytes in memory
- → Virtual memory: each processor/process gets its own linearly addressable bytes in mem.
- → Mapping from physical to VM:
  - Policy: OS determines mapping via page tables
  - mechanism: HW walks the page tables on every memory reference
- → OS + HW enforce process isolation

RECALL: How is the Address space defined?



→ holds procedure activation records

→ memory allocated explicitly by program

→ • statically defined entities
• size determined at link time

→ executable code; fixed size

How does the compiler handle storage layout?

- → Compiler decides where each entity will live at runtime (requires namespace & type info).
- → Assign each named entity a runtime home:
  - a) the logical data area (e.g. stack, static, etc.); depends on lifetime & visibility
  - b) the offset within the data area

What are the storage classes?

- → Automatic, static, irregular
- → Depends on entities lifetime

What are automatic variables? Where are they stored?

- → Lifetime matches that of declaring scope
- → Either the scope's local data area, or a register
- → e.g., if x declared in procedure Fee(), then compiler stores x in Fee()s local data area. x becomes part of Fee()s activation record (AR).
- → If x is local, scalar, unambigous: compiler can store x in a register.

What is an unambigous value?

- → A value that can be accessed by only one name.

What are some examples of ambiguous values?

- → Pointer-based vars
- → call by reference parameters
- → Array-element references
- → Ambiguous values cannot be kept in a register across assignment statements.

What are static variables?

- → Lifetime spans first definition to last use (or, depending on the implementation, lifetime may span entire execution of program)
- → EX: global vars; static vars inside procedures

Where are they stored?

- → Single .static data area per source file

What are irregular variables?

→ Lifetime is under program's control – program code explicitly allocates space (e.g. malloc() in C)

→ Compiler's runtime support library

Where are they stored?

→ The variable itself is stored in the runtime heap

→ The handle (pointer?) to the variable is stored in local or static space.

What are temporary variables?

→ Lifetime is short

→ e.g., x - 2 * y

Where are they stored?

→ Placement depends on size. If variable size is...

Known, and small ⟶ Compiler places in register

large, but bounded ⟶ compiler can place in local data area

unknown ⟶ compiler emits code for runtime heap allocation

– Storage Assignment –

How is storage assigned by the compiler?

1. Assign all small, local, scalar, unambiguous variables to a unique virtual register (reg)
2. Assign each ambiguous valve an offset within its data area.

How are 1D arrays laid out in storage?

→ Let w = width of each element (in bytes); high = max valid index; low = min valid index

→ 1D Arrays laid out as $\boxed{(high - low + 1) * w}$ contiguous bytes of memory

→ Then, the address of V[i] is @V + (i - low) * w

What is one method for laying out 2^+D arrays in storage?

→ Let col = num. columns and row = num. rows

→ Method 1: laid out as col * row * w bytes of contiguous storage

- layout can be row-major or column-major order; language dependent

→ Entire array stored in one big block of memory.

What is the other method?

→ Array of pointers: laid out as one contiguous block of $\boxed{row}$ pointers, each of which points to $\boxed{col * w}$ bytes of contiguous storage

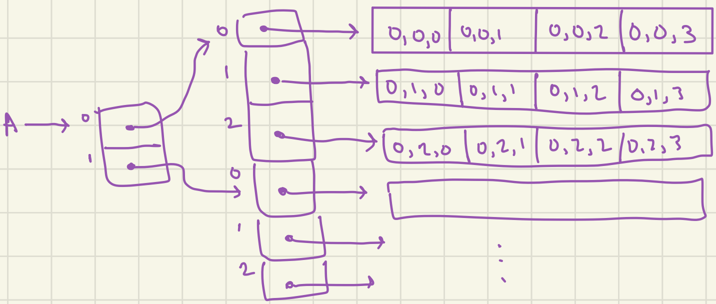
→ E.g., A[i][j][k]. A points to an array of pointers, aka a contiguous block of i elements. Each of these elements is a pointer to an array (aka contiguous block) of j 'row-pointers.' Each of these elements is a pointer that points to an array (contiguous block) of k data elements.

- The array has a total of i * j * k data elements.

→ Total storage needed for the array: $\boxed{(row * ptr) + (row * col * w)}$, where ptr = width of each pointer (in bytes)

Example of storing an array using pointers/indirection vectors?

→ $A[0:1][0:2][0:3]$:



How are strings laid out in storage?

→ Size of layout is not fully determined by length of string

- variable may hold diff strings over its lifetime

→ 2 main storage options:

- Null-terminated: length computation is $O(n)$, where n = length of the string
- Explicit length field: length computation is $O(1)$

How are structures laid out in storage?

→ Space allocated for each field in the struct

→ 2 options for assigning a field's offset within the struct: declaration layout, or compiler controlled

What is an object record? (OR)

→ Data structure / block; each instance of a class has an OR.

→ Contains:

- data members specified by the object's class
- pointer to its class
- pointer to array of the class' methods

→ Additionally, inheritance requires containing data members specified by superclasses & access to superclasses' methods

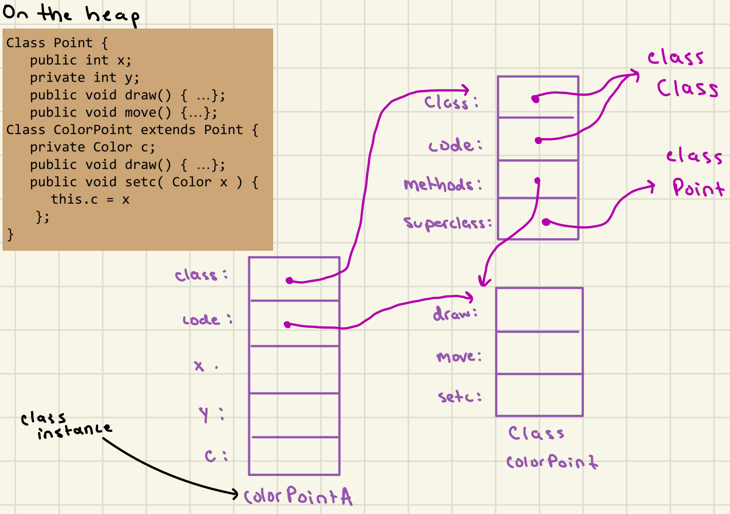
Where are they stored?

→ On the heap

Example?

→

```
Class Point {
   public int x;
   private int y;
   public void draw() { …};
   public void move() {…};
Class ColorPoint extends Point {
   private Color c;
   public void draw() { …};
   public void setc( Color x ) {
      this.c = x
    };
}
```



How are object records laid out in storage?

- → Layout in subclass instances must meet superclass method expectations
- → Single inheritance: prefix layout
- → Multiple inheritance:
  - compiler imposes an order on superclasses
  - subclass instance ORs contain members for all superclasses
  - compiler adjusts OR pointer as needed, when calling superclass' methods

How does the compiler assign storage layout in the IR?

- → e.g. how is storage layout info embedded in the IR
- → Depends on the type of language

How is this done for langs with 'all declarations before executables'?

- → Compiler builds up symbol tables while processing declarations
- → Compiler performs storage layout before processing executables
- → Code gen. uses concrete references

How is this done for languages with 'declare before use'?

- → Compiler builds up symbol tables
- → Code generation uses abstract references
- → compiler performs type inference, storage layout
- → References are refined

What about langs that don't require declarations?

- → Same as for langs that do, but with possible additional passes required

- Alignment & Padding -

What is alignment?

- → Address of a value is divisible by the size of the value (in bytes)
- → ISAs require aligned values, e.g., a 32-bit int begins on a 32-bit addr boundary.

How does the compiler enforce alignment?

- → Ensure data area begins at aligned address
- → Ensure layout of data area is internally aligned

What is the effect of caching?

- → For 2 vars with spatial proximity (declared close together in the code):
  - brought into cache at the same time, on the same cache line
- → For 2 vars with temporal proximity (accessed close tg in time):
  - want them both in cache at the same time
  - non-conflicting cache lines
- → Compilers control depends

Procedures

Ch.6

What is an activation record (AR)?

→ Block of mem. for the control & data values of a procedure instance

- often stack based, e.g. a "stack frame"

→ Key data structure for the implementation of procedures, e.g.: calls & returns, lexically scoped name spaces

What is a procedure?

→ Control-flow based containment of a unit of execution; an abstraction to create a controlled execution environment

→ A callee may be called by many diff caller

→ A Procedure (proc) defines 3 abstractions:

1. call: transfer of control to a proc
2. namespace: a new, protected namespace per call
3. interface: a standardized interaction b/w caller and callee.

How would a proc's SF look?

→

...
arg 10
arg 9
saved registers
local vars
...
Heap

What state is saved across procedure calls?

→ AR pointer (frame pointer, base pointer)

→ Stack pointer

→ RA of caller

→ callee-saves registers

What state is not saved?

→ temporary registers

- Procedure Linkage -

What are the phases of a procedure call?

1. Precall ——— caller
2. Prologue ⎤ callee
3. Epilogue ⎦
4. Postreturn ———

What does the caller do during precall?

→ Save any necessary temporary registers in AR

→ Allocate space for callee's AR

→ Save actual parameter values or references (in registers and/or designated AR slots)

→ Save current AR pointer in AR slot

→ Save address to return to (in dedicated register or AR slot)

→ Jump to callee

What does the callee do during the prologue?

→ Save any needed callee-saves registers :

- return address
- s0 - s11

→ Allocate space for local data area

→ Initialize local variables

RECALL: what are caller- & callee- saves registers?

→ callee-saved: registers that caller expects callee to NOT overwrite. If callee wants to use them, they have to preserve the values in those registers (e.g. on the stack), & then restore them before returning

→ caller-saved: registers that caller expects callee to overwrite. If caller wants to preserve those values, it must save them on stack before making function call.

What does the caller do during epilogue?

→ Save return values (in dedicated AR slot and/or registers)

→ Restore RA

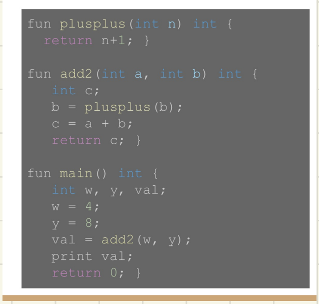
→ Free space for local data area; e.g., move the sp/frame pointer back up

→ Free callee's AR

What does the callee do during postreturn?

→ Restore any needed caller-saves registers

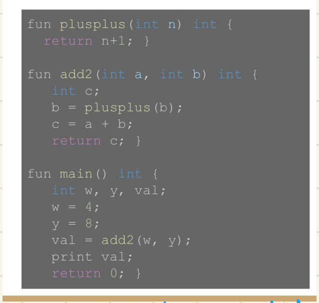
→ Continue execution



```
fun plusplus(int n) int {
  return n+1; }

fun add2(int a, int b) int {
    int c;
    b = plusplus(b);
    c = a + b;
    return c; }

fun main() int {
    int w, y, val;
    w = 4;
    y = 8;
    val = add2(w, y);
    print val;
    return 0; }
```

Example?

→
```
main :
    lw a0 w
    lw a1 y
    jal add2
    jal print
    jal zero exit
  add2 :
   //prologue :
    addi sp sp -12  //make room on stack for RA and params a,b
    sw  ra  0(sp)   // push old RA onto stack
    sw  a0  4(sp)   //push the params a and b on stack
    sw  a1  8(sp)   // b
    lw  a0  8(sp)   //now that param saved, load the param for plusplus in reg
    jal plusplus    //result/ret val will be stored in a0

    lw t0 4(sp)     // t0 = a
    add t0 t0 a0    // t0 = a + b
    addi a0 t0 0    // a0 = t0 = a + b = return val for add2
```



```
fun plusplus(int n) int {
  return n+1; }

fun add2(int a, int b) int {
   int c;
   b = plusplus(b);
   c = a + b;
   return c; }

fun main() int {
   int w, y, val;
   w = 4;
   y = 8;
   val = add2(w, y);
   print val;
   return 0; }
```

(add2 ctd...)

```
//epilogue:
 lw ra 0(sp) //restore old RA
 addi sp sp 12 //restore SP back up
 ret

plusplus: //plusplus makes no calls, so no prologue or epilogue needed
  addi a0 a0 1
  ret
```

Summary of the phases of a proc call?

→ Precall & prologue:

- save state of caller
- set up AR for callee
- transfer params & control to callee

→ Epilogue & postreturn:

- restore state of caller
- tear down AR for callee
- transfer ret vals & control to caller

- Parameter Binding -

What is parameter binding?

→ Mapping caller's actual parameters to callee's formal ones

→ Key to enforcing the procedure abstraction

- proc can be called from multiple call sites
- caller does not need details of callees implementation

→ 2 conventions:

- call-by-value
- call-by-reference

What is call-by-value param binding?

→ Caller copies value into appropriate location – e.g. AR slot, or register

- copy of param val is passed to the callee

→ In the callee, the value has one name: the name of the formal parameter

→ The parameter's value is evaluated & initialized by caller

- Modifications to the param value inside the callee are not visible to the caller

Example?

→

```
int fee (int x) {
  x = x * 2;
  return x; }

int main {
  a = 3;
  b = fee(a); }
```

→

```
main:
  lw a0 a
  jal fee
fee:
  mul a0 a0 2
  ret
```

basically, the caller copies the value of the param into another reg, & then 'passes' on the val via the reg. Upon return, it is caller's responsibility to 'save' the ret value – e.g., by storing it back in memory.

What is call-by-reference param binding?

→ Caller passes the addresses of parameters to the callee (either via a reg or a slot in the callee's AR)

- e.g., rather than lw a0 param1, it would be la a0 param1

→ In callee, value may have more than one name

- the name of the formal param
- the value itself, e.g. if callee has direct access, or if 2 formal params point to the same actual param

→ Changes to the value inside the callee, ARE visible to caller

→ callee has extra level of indirection for each use of the param

Example?

```
int fee (int x, int y) {
    x = 2 * x
    y = x + y;
    return y;
int main() {
    int a = 2;
    int b = 2;
    int c = fee(a,b);
    int d = fee(a,a);
```

→

```
main:
    la a0 a
    la a1 b
    jal fee
    la a1 a
    jal fee
fee:
    lw t0 0(a0)  → callee directly loads in the val
    lw t1 0(a1)
    mul t0 t0 2
    add t1 t0 t1
    sw t0 0(a0)  → callee directly modifies the val
    sw t1 0(a1)
    ret
```

→ In this ex, fee(a,b) will return 8. but fee(a,a) will return 16 - even though both times, we were basically doing fee(2,2).

→ With fee(a,a), both params are pointing to the same param.

How are values returned?

→ Caller sets return value

- in caller's AR
- in designated register

→ Ret val may be a value, or a pointer to a value on the heap.

- Procedures in OO-languages -

How do procedures work for OO langs?

→ AR: manages control & local storage info for methods

→ Object Record:

- Objects' lifetimes are orthogonal to methods' lifetimes
- OR manages persistent state of object
- OR instantiates the inheritance hierarchy

RECALL: What is an object record?

→ A runtime data structure, typically on the heap

→ Pointed to by the Object Record Pointer (ORP)

→ Contains :

- pointer to class
- pointer to class' method vector
- class-specific members

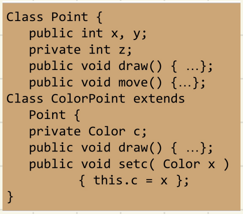
→ OR allocated when an object is created

→ OR deallocated when an object is no longer reachable

Example?

What would the symbol tables look like?

→ Single inheritance, open class structure example:



```
Class Point {
   public int x, y;
   private int z;
   public void draw() { …};
   public void move() {…};
Class ColorPoint extends
   Point {
   private Color c;
   public void draw() { …};
   public void setc( Color x )
         { this.c = x };
}
```

→ RECALL: heirarchical scoping

Class: class
Superclass: None

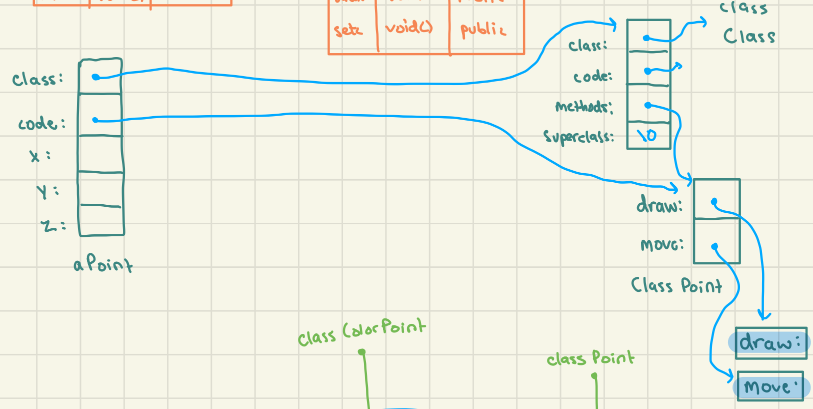
Class: Point
Superclass: Class

x	int	public
y	int	public
z	int	private
draw	void()	public
Move	void()	public

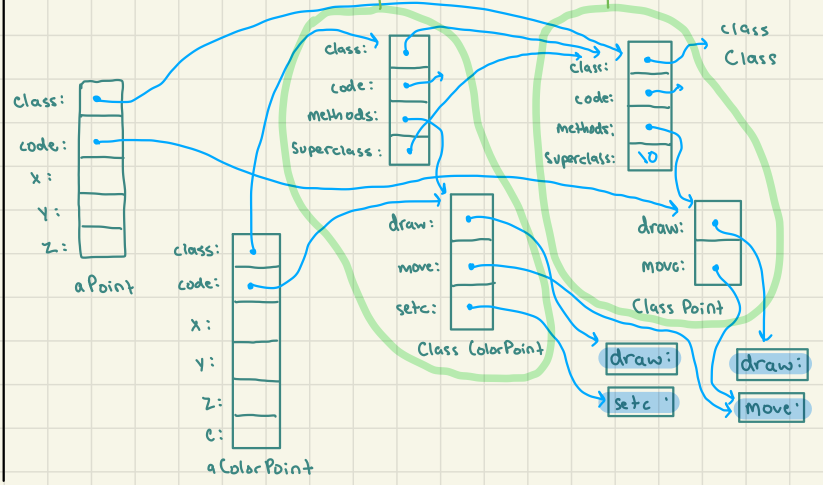
Class: Color Point
Superclass: Point

c	Color	private
draw	void()	Public
setc	void()	public

What would the OR for a Point obj. instance look like?



What if we add a ColorPoint obj. instance?



What is the process of emitting code for method invocation?

→ Invocation happens relative to an obj. instance

→ Instance must be visible at point of invocation

→ Compiler does the following:

- symbol-table lookup to find instance's ORP
  1. lexical hierarchy
  2. class hierarchy
  3. global scope
- emit code to obtain ORP
- emit code to obtain code-vector pointer
- emit code to obtain method pointer
- make procedure call (first param = instance's ORP)

What is a closed class structure?

→ Method impl. is known at compile time

→ ST lookup produces the label of the method

→ compiler emits code to generate the proc call

- Addressing Variables within the Procedure -

How are variables addressed?

→ 2 parts: base addr, and offset

→ local vars & static + global vars have static base addresses

→ Variables with dynamic base addresses:

- those belonging to enclosing lexical scope
- heap-allocated objects

Example of a static base address?

```
.data
   w: .word 4
   y: .word 8

main:
   lw a0 w
   lw a1 y
```

static vars

uses the compiler-generated label

How does the compiler handle dynamic base addresses?

→ For dynamic vars, base addr. is NOT known at compile time

→ Compiler does the following:

1) ST lookup to Find variable's coordinates (lexical hierarchy, global scope)
2) emit code to obtain correct ARP
3) emit code to add offset to pointed-to AR
4) emit code to load the variable @ the calculated address

Code Shape

Ch.7

What is code generation?

→ Mapping source-language constructs to the target machine's instruction set

- * Expression evaluation
- * Accessing vars & aggregate data structures
- * Control-flow constructs
- * procedure calls

What is the register-to-register model?

→ emit code using virtual registers for unambiguous, scalar values

→ later: register allocation assigns virtual regs to physical ones

What is the memory-to-memory model?

→ emit code using memory as primary home for all values

→ later: optimizations to increase register use

What is the emitted code?

→ 3-addr code IR ; array of tuples

What is code shape?

→ The explicit & derived knowledge about the source prog., encoded in the IR

- live values
- registers in use
- memory references
- type information
- instruction mapping

→ For ex, consider x+y+z ... code shape determines later opportunities for optimization

→ Deciding code shape often requires 2 passes:

1. learn the surrounding context
2. decide code shape & emit code

– Expression Evaluation –

RECALL: What is an expression?

→ e.g. x+y+z

→ Represent a significant portion of code

→ Efficient evaluation improves runtime of overall program

→ Target machine with 3-addr form operators

- allows compiler to name the result of operations for later reuse
- most RISC ISAs, including RISC-V

How is code generated for expressions?

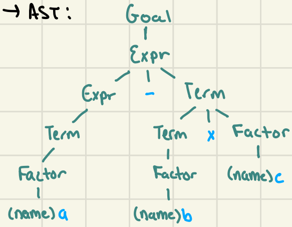
→ Walk AST to: emit code to put operands in registers, emit code for expressions

What would the program to generate code for expressions look like?

```
expr(node) {
   int result, t1, t2
    switch (type(node)) {
       case x, /, +, - :
            t1 = expr(node.left_child)
            t2 = expr(node.right_child)
            result = next_regrster()
            emit( node.op, t1, t2, result)
            break
       case NAME :
            result = access_value(node.st_lookup())
            break
       cese NUM :
            result = access_number(node.st_lookup())
            break
     } return result; }
```

→ Ex: a-b×c

→ AST:



Example walkthrough?

→ For a-b×c, with a, b, c on the stack:

```
lw   t0  0(sp)    ⟵ t0 = a
lw   t1  4(sp)    ⟵ t1 = b
lw   t2  8(sp)    ⟵ t2 = c
mul  t3  t1  t2   ⟵ t3 = b×c
sub  t4  t0  t3   ⟵ t4 = a-b×c
```

What is the convention for register usage in RISC-V?

Reg Name	Reg Num	Description
zero	$0	constant 0
ra	$1	return address
sp	$2	stack pointer
t0-t6	$5-7, $28-31	temporaries (caller-saves)
s0-s11	$8-9, $18-27	callee-saves (safe for caller)
a0-a1	$10-11	function args & return valves
a2-a7	$12-17	function args

→ RECALL COMP 311

How does the compiler reduce register pressure?

- → In a binary operation, the operand requiring more registers should be computed first
  - results in the same number of virtual registers (VRegs)
  - later, reg. allocation will produce code using fewer registers
- → Requires 2 passes by compiler:
  1. compute (virtual) register demand for each operand
  2. decide order of subexpression evaluation & emit code

How are expression operands handled?

- → Variables:
  - In memory (AR ; labeled area in .data)
  - In (virtual) register
- → Constants:
  - labeled area in .data
  - Immediate values
- → Temporary values (aka the result of a subexpression): in a register
- → Procedure calls:
  - Postreturn sequence puts result in appropriate location
  - Subexpression evaluation order cannot be reordered around procedure calls

What about for mixed-type operands?

- → Source languages allow some mixed-type expressions (eg int + float)
- → Compiler converts operand(s) to the appropriate type
- → In compiler-time conversion, compiler looks up conversion rules & applies them at the point of emitting code
- → In Run-time conversion:
  - compiler adds type tag to value
  - runtime case analysis jumps to appropriate subroutine for mixed types

How does the compiler handle assignments?

- → e.g. a = b + c
- → For RHS (eg b+c): expression evaluation; result is a value ; compiler emits code
- → For LHS (eg a): evaluation to access a value
  - result is a location (eg addr where a is stored)
  - compiler emits code to move RHS value to LHS location

- Accessing Values -

How are values kept in registers accessed?

- → RECALC: values kept primarily in registers: scalar, unambiguous, & temporary values
- → IR may assign register-based values to virtual registers
- → Registers can be read & written to in the same cycle
- → Ideal: values kept in registers wherever they are used frequently
  - optimization, register allocation

How are values kept in memory accessed?	→ RECALL: values kept primarily in memory: • ambiguous names • names that have global visibility • names that have static lifetimes → IR represents the name with its coordinates • lexicel level (e.g. global = 0), & offset → Type info in the IR is needed for emitting correct operations
Which values are kept on the stack?	→ Local values → Spill values from registers → Storage allocated in the activation record (AR) → These values are referenced by offset (to ARP or SP)
How are variables of surrounding scopes accessed?	→ E.g., values from enclosing lexical scopes (excluding the global scope) → IR represents the name with its coordinates (lexical level, offset) → compiler emits code to traverse the chain of scopes
How are static & global vars accessed?	→ compiler emits code to access using the label
How are parameters accessed?	→ Compiler emits code according to the ISAs linkage convention
How are values on the heap accessed?	→ Virtual address must be maintained • in a register • on the stack • as a field in another heap-based entity

- Accessing Aggregate Data Structures -

What are the memory access instructions in RV32I?	→ load word: lw rd imm(rs1) ... reg[rd] = mem[reg[rs1] + imm] • loads 4 bytes (16 bits) • Immediate is 12 bits, sign-extended → load word pseudo-inst: lw rd label → load byte: lb rd imm(rs1) ... reg[rd] = mem[reg[rs1] + imm] • loads 1 byte, sign-extended → load byte unsigned: lbu rd imm(rs1)... same as lb but loaded in byte is zero-extended → Store word: sw rs2 imm(rs1) → Store byte: sb rs2 imm(rs1) ... stores LSB of rs2
RECALL: What are aggregate objects?	→ Structs, class objects, vectors, multi-dimensional arrays
How are they accessed?	→ Accessing indiv. elements of the aggregate: accessible via start addr of the object + offset within the aggregate → Elements have individual type information

How are struct elements accessed?

→ EX: Struct Ex { int x; int y; }

1. ST lookup to find start address
   - struct's storage class determines location: local AR, enclosing scope, static/global label, or heap
2. Emit code to put start address in register, eg la t0 struct1
3. ST lookup to obtain element offset & type
4. Emit code to add offset to start address, eg addi t0 t0 4, for int y
5. Emit load operation(s) ... use type info to select appropriate instructions

How are object members accessed?

1. ST lookup to find start address
   - Object record pointer (ORP) is associated with the object's source-language name
2. Emit code to put ORP in register
3. ST lookup to obtain object member's offset & type
4. Emit code to add offset to OR's start address
5. Emit load operation(s) ... use type info to select appropriate instructions

How are vector elements accessed?

1. ST lookup to find start address & element type
   - vector's storage class determines location: local AR, enclosing scope, static/global label, or heap
2. Emit code to put start address in register
3. Emit code to compute offset in vector
4. Emit code to add offset to start address
5. Emit load operation(s) ... use type info to select appropriate instructions

Example for vector elements?

→
```
# vector [0:9]... a0 holds the address
# of V and a1 holds the index i
addi t1 zero 4     → move length of each el. into t1
mul t0 a1 t1       → t0 = index * size of el.
add a0 a0 t0       → a0 = base addr + offset
lw t0 0(a0)
```

How are multidimensional arrays accessed?

→ RECALL: 2 ways to store: row/col-major order, OR indirection vectors

for 2-D arrays

→ Num. of mem. accesses for each type is different; indirection vectors can be slow
   - Mem. access is expensive compared to calculations
   - May be harder for compiler to take advantage of locality

from 'Storage Layout' lec

RECALL: What are indirection vectors?

→ Innermost level is a set of vectors containing data elements

→ Outer level(s) are a (set of) vectors containing pointers to the next inner level

How are arrays as parameter arguments handled?

- → Different call sites might pass arrays with different dimensions
- → Callee needs dimension information
- → Passed via a Dope vector, which is a fixed size & layout and contains a descriptor for the array

What are range checks?

- → Test that the given array index falls within the allocated range
- → Range checking is determined by the language design... either:
  - compiler does check statically
  - compiler builds code to check dynamically
  - compiler does no checking (e.g. C ... there are buffer overflow & other security vulnerabilities)

– Boolean & Relational Operators –

How are boolean & relational operators used?

- → For specifying **control flow**, e.g. conditionals & loops
- → HW support: assembly instructions for comparing, performing boolean ops (e.g. AND, OR), and acting upon the result (e.g. beq, bne, blt, bne)

What is short-circuit evaluation?

- → Evaluating a boolean expression only until the result is determined
  - e.g. for if (a<b) && (c<d) ... if (a<b) evaluates to false, then (c<d) won't even be evaluated
  - If operand is false/true for an AND/OR operation, 2nd operand doesn't need to be evaluated
- → Evaluate the operand that requires fewer operations first - aka, the more shallow subtree in an AST

– Control-Flow Constructs –

What is a basic block?

- → DEFN: Maximal-length sequence of straight-line, unlabeled, unpredicated instructions
- → **Control-flow transfer ends the block**
- → Labeled statement ends the block (assuming label is a possible target branch)
- → Block starts with labeled statements & the fall-thru instruction of a branch

How is an if-then-else section constructed in assembly?

- → Build the basic blocks (if block, else block), then insert instructions to implement the control flow.
- → E.g.:

```
if (condition) {
    block_1
} else { block_2 }
} block_3
```

→

```
La:
  b{eq, ne, ge, lt} rs1 rs2 Lb
  # code for block 1
  jal zero Lc
Lb:
  # code for block 2
Lc:
  # code for block 3
```

statement that checks for OPPOSITE of condition; if condition false, jump to block$_2$. Else, execute block$_1$.

What logical & branching instructions does RV32I have?

→ slt, slti, beq, bne, bge, blt

→ jump and link: jal rd label ... sets rd = PC + 4, then sets PC = PC + offset (eg offset given by label) ... label is 20 bits

What instructions can be used to avoid branching?

→ Sample Code:

```
if (a < b) {
    x = c + d }
else { x = e + f }
```

→ NOT in RISC-V, but:

- conditional move operations: move the value in one of 2 registers into dest reg, based on condition. e.g., for sample code above:

```
add t0 rc rd
add t1 re rf
slt t2 ra rb
mvcon rx t2 t0 t1
```

- predicated operations: instr. executes or not, based on the condition, e.g.:

```
slt t0 ra rb
xori t1 t0 0x01
addeqz t1 rx rc rd
addeqz t0 rx re rf
```

What are condition codes?

→ Provide detailed results from comparison instructions

→ E.g., in x86:
- zero flag: result is 0
- sign flag: result is negative
- carry flag: a borrow was needed (unsigned)
- overflow flag: an overflow occurred (signed)

→ Enables a variety of branch instructions

What are loops (high level)?

→ Perform iteration, w/ common features being:

- Initialization
- conditional
- update state used in guard

→ In the basic case (eg no nested loops): one initial branch + one branch per iteration

How are for loops implemented in assembly?

→ e.g.

```
for (expr1; expr2; expr3) {
    // loop body }
    // post loop
```

1. evaluate expr1
2. if NOT (expr2) go to no.5
3. loop body
4. evaluate expr3. if expr2 go to no.3
5. post-loop

How are while loops implemented?

→ Similar structure to for loops

→ condition expression, but no initialization

→ double test loops are better

What are until loops?

→ Guard is at the end of the loop body (execute until guard is true)

→ Execute at least once

→ More compact than while loop

What is a tail call?

→ A call that is the last statement in a procedure

What is tail recursion?

→ A recursive call, that is a tail call

→ Optimizations allow eliminating much of the procedure set-up & tear-down w/ tail recursion for iteration ... allowing resulting code to be as efficient as traditional loop

How are break statements handled?

→ Compiler emits immediate jump to labeled breakpoint, or labeled stmt following the loop or case

How are skip/continue statements handled?

→ Compiler emits immediate jump to code evaluating the loop condition

How are case statements handled?

→ Basic case: evaluate the controlling expr, branch to the selected case, execute the code for that case

→ Break stmts exit the case. Otherwise, end of exec of a case falls through to the next case

→ Challenge: efficiently branching to the selected case

- Compiler uses value of controlling expr to locate the corresponding case

What are the methods for efficiently branching to the selected case?

→ Linear Search

→ Computed address

→ Binary Search

What is linear search?

→ Implement the case as a series of nested if-then-else statements

→ Cost of branching to correct case depends on order of cases

- Compiler should order cases according to estimated execution frequency

→ Works well for small number of cases

What is computed address?

→ Compiler builds a jump table, indexed by case: vector of block-address labels

→ Efficient when case labels form a compact set

What is binary search?

→ Compiler builds ordered table of case labels + block-address labels

→ Compiler emits binary search:

- Find the right label
- branch to associated block addr

→ Useful when: num. of cases is small and/or label set is sparse

How are String operations handled?

→ String length: null-terminated storage, or explicit length storage

→ String assignment: HW support for byte-oriented mem. operations is useful

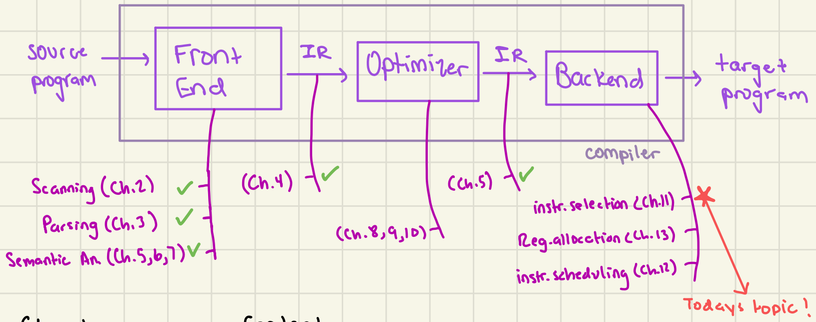
→ String concatenation:

- appending = string length + assignment
- new string = 2 assignments

Instruction Selection

Ch.11

RECAP: Where are we now?



Chapter	Content
5	Namespaces, Naming Env., Type Info, SDT, Storage Layout
6	Procedures
7	Code Shape

What is the role of the backend?

→ Optimizer has primary responsibility for efficiency

→ Backend also plays a role:

- * Instruction Selection: local mapping from optimized IR to ASM
- * Reg. Allocation: mapping virtual Registers to ISA ones
- * Inst. Scheduling: scheduling of ISA Inst.s on target machine (TM) functional units

What is instruction selection?

→ Rewrite IR code into TM's ASM (assembly)

→ Challenge: many possible ASM implementations for each IR statement

→ Goal: efficient, local rewriting

→ Ex: x = 1 / y = 1 could turn into addi t0 $0 1 / add t1 t0 $0 or add t0 $0 1 / slt t1 t0 $0 , . . . etc.

What is the instruction selector (I.S.) generator?

→ Aka backend generator, code-generator generator

→ Compiler writer produces specifications: IR, ISA, & mapping b/w them

→ Then, backend generator (BG) builds inst. selection engine for the given specs (similar to how parser generators work)

→ Basically, instruction selection must take IR representation(s) of a piece of code—like an AST and/or ST—& produce assembly language program.

What is our running example?

→ a = b − 2 × c, where: a = local var in AR, w/ base addr = ARP & offset = 4
b = call-by-ref var in AR, w/ base addr = ARP & offset = −16
c = global var, w/ base addr = label "GL" & offset = 12

What would its IRs look like?

What is our running example?

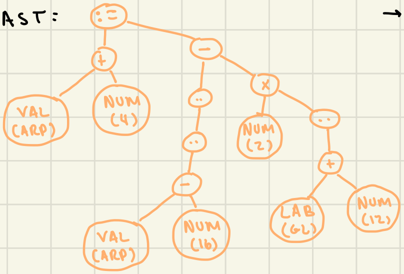
→ a = b-2×c ,where: a=local var in AR ,w/ base addr = ARP & offset = 4

b = call-by-ref var in AR, w/ base addr = ARP & offset = -16

c = global var, w/ base addr = label "GL" & offset = 12

What would its IRs look like?

→ AST:



→ Table of quadruples:

Result	Arg_1	Arg_2	Op
t0	2	c	×
a	b	t0	-

How would the I.S. use these IRs?

→ Linear IR (e.g. quadruples):

- text-based pattern matching
- based on peephole optimization

→ Tree-based IR (eg AST): pattern matching on trees

-Peephole Optimization-

What is peephole optimization?

→ One approach to inst. selection

→ Using pattern matching over short inst. sequences to find local improvement

→ Traditionally compiler's last pass - aka, reading & writing ASM

→ Idea: 1) Translate IR into lower-level IR (LLIR)

2) Systematically improve LLIR via peephole optimization

3) Map optimized LLIR to TM instructions (ASM)

→ Basically, the compiler examines the code in small adjacent /overlapping sequences/ windows. In a given window (small sequence of code), it examines the operations in this window, looking for specific patterns that it can improve. When it recognizes a pattern, it rewrites it w/ a better instruction sequence.

Examples of peephole optimizations? (PpO)

→ EX: a store & then load of the same value:

```
sw t0 8(sp)
lw t0 8(sp)
```

→ P.O. can replace the load w/ a copy inst., which is less expensive:

```
sw t0 8(sp)
mv t1 t0
```

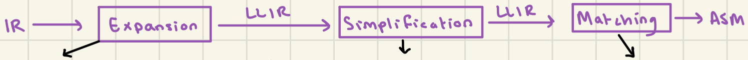
→ Ex: unnecessary branching:

```
jal zero La
jal zero Lb
```

→

```
jal zero Lb
```

What are the phases of PpO?



- Rewrite IR into LLIR
- Capture all the IR effects
- Template driven
- Bottom-up rewrite (calculate liveness information)

- 'Sliding window' of 2-4 instructions
- Top-down rewrite:
  - forward substitution
  - algebraic simplification
  - evaluation of constant-valued exprs
  - elimination of dead effects

- Library of patterns
- Top-down rewrite
- Final ASM (that may use virtual registers):
  - preserves all vals in LLIR (mem, regs, condition codes)

- IS PpO: Expansion -

What does the IS do during expansion?

→ Rewrite IR into an LLIR, capturing all memory details ... Bottom-up rewrite

→ Template-driven: Expander uses a template for each IR op, & substitutes appropriate reg names, constants, & labels in the template.

→ Expander rewrites IR, op by op, into a sequence of LLIR ops that represent all the direct effects of an op.

Example of the LLIR that expansion might produce?

→ GIVEN: The linear IR for $\boxed{a = b - 2 \times c}$:

Result	Arg_1	Arg_2	Op
t0	2	c	×
a	b	t0	−

→ Resulting LLIR:

load c

$r_{10} \leftarrow @CP$

$r_{11} \leftarrow @GL$

$r_{12} \leftarrow r_{10} + r_{11}$

$r_{13} \leftarrow Mem(r_{12})$

$r_{14} \leftarrow 12$

$r_{15} \leftarrow r_{14} + r_{13}$

$r_{16} \leftarrow Mem(r_{15})$

obtain base addr for c, add offset, load val @ this addr

$r_{17} \leftarrow 2$

$r_{18} \leftarrow r_{17} \times r_{16}$ → $r_{18} = 2 \times c$

load b

$r_{19} \leftarrow 16$

$r_{20} \leftarrow r_{ARP} - r_{19}$

$r_{21} \leftarrow Mem(r_{20})$

$r_{22} \leftarrow Mem(r_{21})$

obtain base addr for b, add offset, load val @ this addr

store a

$r_{23} \leftarrow r_{22} - r_{18}$ → $r_{23} = b - (2 \times c)$

$r_{24} \leftarrow 4$

$r_{25} \leftarrow r_{ARP} + r_{24}$

$Mem(r_{25}) \leftarrow r_{23}$

store res at a's loc. in memory

What is 'last use'?

→ A reference to a name (eg a reg holding a certain var/value) after which the value represented by that name is no longer 'live' (used)

→ Simplifier needs expansion to "mark last use". For ex, with

```
li t0 2
add t1 t0 t0
```

, we could easily simplify this to `addi t1 zero 4`. BUT, expander cannot eliminate the first op. unless it knows that t0 is not live after its original use in the second op.

How does the expander mark last use?

→ Compute LiveOut sets for each block; then, in a backwards pass over the block, track which vals are live at each operation

→ Then, walk each block from bottom-up & build the LiveNow set: set of values that are live at each operation.

How is the LiveNow set computed?

→ Initialize LiveNow with either all of the names used in more than one block, or the LiveOut set for the given starting block.

→ Then, while processing each operation $r_i \leftarrow r_j \text{ op } r_k$, update LiveNow by:

1) removing r_i from it
2) adding r_j and r_k to it

→ RESULT: alg produces, at each step, a LiveNow set that is as precise as the initial info used at the bottom of the block.

–IS PpO: Simplification–

What does the IS do during simplification?

→ Sliding window: considers 3-4 instructions at a time

→ Pattern-match to identify simplifications:

- forward substitution
- evaluation of constant-valued expressions
- algebraic simplification
- elimination of dead effects

→ Emit simplified instructions; discard dead instructions

→ Basically, simplifier makes a pass over the LLIR, examining the ops in a small window on the LLIR & systematically trying to improve them.

What is forward substitution?

→ Substituting defined values into uses later in the window

→ Example:

```
la   r5   L1
li   r6   12
add  r7   r5 r6
lw   r8   0(r7)
```

→

```
la   r5   L1
lw   r8   12(r5)
```

→ Rather than defining (12 + L1) as a new value, we can directly do a load word.

What is algebraic simplification?

→ Using algebraic identities to simplify, e.g.:

```
subi t0 t1 0
mul  t3 t0 t2
```

→

```
mul t3 t1 t2
```

→ Algebraic simplifications to look for:

- $a + 0 = a$
- $a - 0 = a$
- $a - a = 0$
- $a \times 0 = 0$
- $a \times 1 = a$
- $a \times 2 = a + a$
- $a / 1 = a$
- $a / a = 1$
- $a \&\& a = a$
- $a \,||\, a = a$
- $a >> 0 = a$
- $a << 0 = a$
- $a << 1 = a + a$

What is constant folding?

→ Evaluate constant-valued expressions & substitute the value

```
li r1 2
add r2 r1 r1
```

→

```
addi r2 zero 4
```

How does the simplifier eliminate dead operations?

→ Simplifier can only eliminate defining operations if the defined value is dead

→ Simplifier uses the LiveNow computation from the Expander

Example?

```
                 LN: ∅
li r1 2
                 LN: {r1}
add r2 r1 r1
                 LN: {r2}
```

What woold oor example LLIR look like after simplification?

```
r10 ← @CP
r11 ← @GL
r12 ← r10 + r11
r13 ← Mem(r12)
r14 ← 12
r15 ← r14 + r13
r16 ← Mem(r15)
r17 ← 2
r18 ← r17 × r16
r19 ← 16
r20 ← rARP − r19
r21 ← Mem(r20)
r22 ← Mem(r21)
r23 ← r22 − r18
r24 ← 4
r25 ← rARP + r24
Mem(r25) ← r23
```

→

```
            r13 ← Mem(@CP + @G)
            r16 ← Mem(r13 + 12)
            r18 ← r16 + r16
            r21 ← Mem(rARP − 16)
            r22 ← Mem(r21)
            r23 ← r22 − r18
Mem(rARP + 4) ← r23
```

– IS P_PO: Matching –

How does matching work?

→ Matcher compares simplified LLIR against the pattern library, looking for the pattern that best captures all the effects in the LLIR

→ Some operations are a one-to-one match; others may require multiple ISA instructions for one LLIR inst. (this may introduce temporary names)

→ Matcher must know:

- size of constants
- address modes
- available memory operations

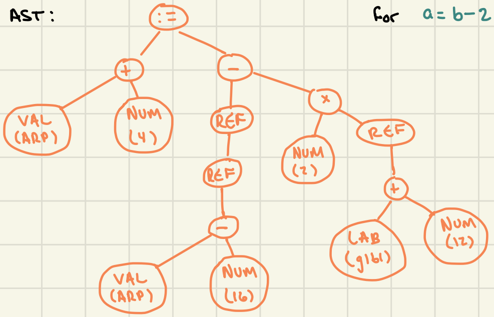
– IS via Tree-Pattern Matching –

RECAP ?

→ Peephole Optimizations use linear IRs to do instruction selection

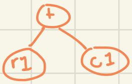
→ Another way to do IS is using tree-based IRs, like ASTs (which can be an LLIR)

→ Example AST: for $a = b - 2 \times c$



What is an operation tree ?

→ Tree-representations of operations in target ASM . e.g. :



add r1, c1 => r3

What is the library of tree patterns?

→ Rewrite rules; each pattern maps constructs in the IR to operations in target ASM

→ Each pattern contains:

- low-level IR pattern tree
- ASM code template
- cost

What is tiling?

→ a Tile is <x, y>, where:

- x is a subtree in the AST
- y is a tree pattern from the Library of Tree Patterns
- pattern y can implement the subtree at node x

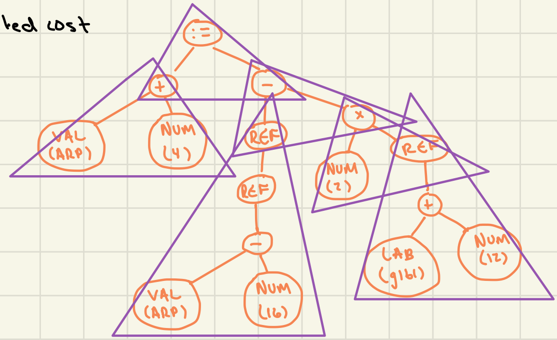
→ Tiling: a set of tiles that implement the AST

- covers every node in the AST
- overlap between any 2 trees occurs at a single node
- where 2 pattern trees overlap, they agree in storage class & valve type
- root of each pattern tree overlaps a leaf of another pattern tree, unless it covers the root of the AST

What is the goal?

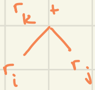
→ Find a locally optimal tiling

→ Triangles represent tree patterns: each has an associated ASM code template & associated cost



How do we describe an LLIR AST using prefix notation?

→ Write the op. tree for add, for ex, as:



$\longrightarrow + (r_i, r_j)$

→ Combine them all. Ex for the AST above:

:= (+ (VAL(ARP), 4) , − (− (VAL(ARP), 16) , × (2, + (LAB(g161), 12)))

Register Allocation

How does register allocation work?

1. Determine, at each point in the code.
   - which valves reside in registers
   - which valves reside in memory
2. Rewrite the code to implement the allocation
   - spill & restore valves when there are too many live valves
   - Keep ambiguous values in memory

RECALL: What are registers?

→ Defined by the ISA

→ Provide fast access b/c physically on the CPU, and no address translation

What is register allocation?

→ Input: IR program that is <u>nearly</u> executable ASM, but uses an arbitrary number of virtual registers

→ Output: equivalent program that fits into the ISA's register set

- aka executable ASM

→ Can be followed by:

- instruction scheduling

RECALL: What is the register-to-register model?

→ IR uses as many virtual registers (VRs) as needed

→ Register allocator maps VRs to physical registers (PRs)

What is the memory-to-memory model?

→ IR establishes a home memory location for all named valves

→ Register allocator can promote a valve to a register for a longer range of its lifetime

What are 2 methods for register allocation?

→ Local register allocation:

- each basic block analyzed independently (scope of analysis = basic block)
- a single class of physical registers (PR) is given
- assumes analysis begins with no valves in PRs
- namespace for valves: Live Range
- key algorithm: compute most distant next use to determine spills

→ Global register allocation:

- each procedure is analyzed as a whole
- may be multiple classes of physical registers
- Key algorithm: uses graph coloring to compute allocation
- namespace for values: Live Range

How does local register allocation work?

1. Build a new namespace of Live Ranges (LR)
2. Perform register allocation (calculate interference; generate spill code)
3. Rewrite the code using the namespace of physical registers

What are Live Ranges? (LR)

→ DEFN: A closed set of related definitions and uses that serves as the base name space for register allocation

→ Each LR corresponds to a single valve

- from the value's definition
- to the value's last use

→ In a basic block, each LR

- starts with one definition
- extends until that definition's last use

→ An LR starts: with the first operation after it is defined

- differentiates b/w a use and a definition in the same operation

→ An LR ends:

- at last use of the name, OR
- when the name gets a new value

What is liveness?

→ A variable v is live at point p if it has been defined along a path from the procedure's entry to p, and there exists a path from p to a use of v along which v is not redefined.

→ Anywhere that v is live, its value must be preserved because subsequent execution might use v.

Example?

→ For $a = a \times 2 \times b \times c \times d$, we have this IR block, and each distinct live range in the block:

1 $r_{arp} \leftarrow Mem(ARP)$

2 $r_a \leftarrow Mem(r_{arp} + offset)$

3 $r_2 \leftarrow 2$

4 $r_b \leftarrow Mem(B)$

5 $r_c \leftarrow Mem(C)$

6 $r_d \leftarrow Mem(D)$

7 $r_a \leftarrow r_a \times r_2$

8 $r_a \leftarrow r_a \times r_b$

9 $r_a \leftarrow r_a \times r_c$

10 $r_a \leftarrow r_a \times r_d$

11 $Mem(A) \leftarrow r_a$

Register	Interval
r_{arp}	[1,11]
r_a	[2,7]
r_a	[7,8]
r_a	[8,9]
r_a	[9,10]
r_a	[10,11]
r_2	[3,7]
r_b	[4,8]
r_c	[5,9]
r_d	[6,10]

→ basically, a reg's LR is the interval over which it represents (& is used as) a given velve. E.g., r_a has an LR from [2,7] b/c line 7 uses the val from operation 2. Op. 8 uses the valve from Op.7 (rather than op.2), SO, r_a has an diff LR from [7,8]

More about LRs?

→ A reg. allocator doesn't have to keep each distinct LR for one register, e.g. r_a in the same reg... it can treat each LR in the block as an independent val for allocation & assignment. For ex, the ILOC ex from prev. page could become:

```
lw    ra   @a(sp)
addi  r2   nero 2
lw    rb   @b(sp)
lw    rc   @c(sp)
lw    rd   @d(sp)
mul   r1   ra  r2
mul   r3   rb  r1
mul   r4   r3  rc
mul   r5   r4  r2
sw    r5   @a(sp)
```

What is interference?

→ 2 LRs interfere if there exists an operation for which both are live and they (probably) don't have the same value.

in local RegAlloc, 2 LRs interfere if their spans overlap

→ Two LRs can share a physical reg. i.f.f. they do not interfere, and use the same PR set... 2 LRs using distinct PRs cannot interfere

- in local RegAlloc, all LRs use the same PR set

– Local Register Allocation –

What is Best's Algorithm?

1. Compute live information (bottom-up pass)
2. Add interference info (top-down pass)

→ Idea: Most distant next use

- spill (aka store in mem) the PR with the val whose next use is furthest in the future
- reduce demand for PRs over the longest interval

→ Optimal choice if spill costs are uniform (in general tho this isn't the case)

What are non-uniform spill costs?

→ Dirty values, clean values, rematerializable values

What is a dirty value?

→ A value that does not exist in memory

- STRAT: store at spill points (out of registers (?)), load at restore points

What is a clean value?

→ A value that is already in memory

- STRATEGY: load at restore points

What is a rematerializable value?

→ A value that can be recomputed at each use

- STRATEGY: recompute at restore points (e.g. load immediate operations)

How are the live ranges determined?

→ Maintain 2 maps:

- VRtoLR: initialized to INV for all virtual registers (VRs)
  - → VRtoLR[x] = y means that the curr. val in register x belongs to LR y.
- PrevUse: initialized to inf for all VRs
  - → maps virtual register to index of closest future use
  - → PrevUse[v] = the next inst. that uses VR v (while scanning bottom-up).

→ Bottom-up Pass

- visit definitions first, then uses
- update maps
- Fill in the LR and Next Use (NU) fields

What would this pseudocode look like?

```
For each operand of the inst.:
    For each op defined:
        if VRtoLR[op.VR] == INV, then VRtoLR[op.VR] := next LR
        op.LR = VRtoLR[op.VR]
        op.NU = PrevUse[op.VR]
        PrevUse[op.VR] = inf
        VRtoLR[op.VR] = INV
    For each op used:
        if VRtoLR[op.VR] == INV, then VRtoLR[op.VR] := next LR
        op.LR = VRtoLR[op.VR]
        op.NU = PrevUse[op.VR]
    For each op used:
        PrevUse[op.VR] = index of current instruction
```

So how are registers assigned?

→ Process uses before definitions

→ For each operation:

1. For each use, look for an existing PR assigned to the LR
   - a. maintain LRtoPR map
   - b. call getPR if new PR is needed
2. Determine if a use is the last use of the LR
   - a. if so, free the PR, making it avail. for reassignment
3. For each definition, assign a PR to the LR
   - a. call getPR
   - b. update the LRtoPR map

How would the getPR function work?

1. Find a PR for a reference v
2. If there exist free PRs, use the next available (maintain a stack of free PRs)
3. Else, choose an LR to evict from its PR:
   - spill the LR val to mem/stack
   - reassign the PR to v
   - if the ref to v is a use rather than a definition, restore its valve from its mem. location, to the newly assigned PR

RECAP: What are the steps for RegAlloc?

1) Translate to namespace of live ranges
2) Identify interferences b/w live ranges
3) Find an allocation
4) Decide what to spill

What is the "Max Live" valve?

→ Maximum demand for registers in a basic block

→ Given by the max. num. of concurrently live ranges

→ If Max Live is greater than the num. of PRs, a spill is needed

– Global RA –

What are the challenges with local RA?

→ LRs have multiple definitions & uses

→ RA must coordinate across basic blocks

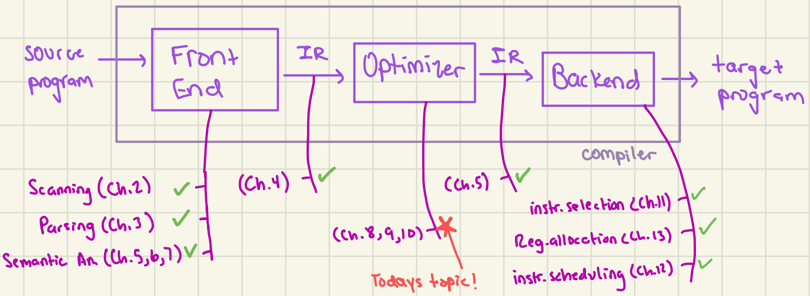
→ Spill cost estimation must account for blocks that execute multiple times

How does global RA work?

1. Find LRs
2. Build interference graph
3. Coalesce copies – loop to step 2
4. Estimate spill costs
5. Find a coloring
6. Insert spills – loop back to step 1

Optimization

RECAP: Where are we now?



What are the goals & process in optimization?

- → Goal: improve performance of compiled code
- → Process: Analysis, Transformation

What are the considerations?

- → Safety: transformed code produces same results as original code
- → Profitability: (trnsfm) transformation improves performance
- → Risk: Trnsfms intended to improve performance can make it harder for the compiler to generate good code

What are the different scopes where we can perform optimization?

- → Local: one basic block
- → Regional: a subset of the CFG
- → Global: one procedure
- → Interprocedural: 2 or more procs

- Local Value Numbering -

What is the "local scope" exactly?

- → A single basic block = scope for local OPT methods; so each basic block

RECALL: What is a basic block?

- → RECALL: basic block = maximal-length sequence of branch-free code
  - Starts w/ a label or fall-through instruction
  - Ends with a branch or jump
  - Instructions execute sequentially
  - If one inst. in a BB executes, they all do

What is Local Value Numbering (LVN)?

- → A type of local optimization for eliminating redundancy
- → Finds & eliminates redundantly computed expressions:
  - exprs previously computed in the basic block, AND
  - exprs where no operands have been redefined since 1st computation

Examples of redundant expressions?

→

```
a ← b + c
b ← a - d
c ← b + c
d ← a - d
```

c ← b + c → repeats line 1

d ← a - d → repeats line 2

```
a ← b + c
d ← b
e ← d + c
```

L2 is redundant b/c we could just do "e ← b + c"

What is the algorithm for LVN?

- → Traverse block top-to-bottom, and map each value to a distinct number (each "value" ≈ each operand or expression)
- → Two vals map to same number i.f.f. they are equal in all possible executions
- → Use a hash of each val to index into the mapping table (or use a dictionary)

What is the pseudocode?

- → Start with an empty hash table mapping value names to their distinct numbers / IDs :

HT =

Val number	name
. . .	. . .

```
num = 0
For each instruction T ← L Op R :
    $VN_L$ = getNumber (L)
    $VN_R$ = getNumber (R)
    if < $VN_L$, Op, $VN_R$ > is an entry in HT :
        x = getName (< $VN_L$, Op, $VN_R$ >)
        $VN_x$ = getNumber (< $VN_L$, Op, $VN_R$ >)
        if getNumber (x) == $VN_x$ :
            rewrite instruction to be "T ← x"
        else :
            set the "name" associated w/ the same
            value, < $VN_L$, Op, $VN_R$ > (aka name whose
            value = value for < $VN_L$, Op, $VN_R$ > entry)
            to be INVALID
    else :
        HT [< $VN_L$, Op, $VN_R$ >] = num
        num += 1
    $VN_T$ = HT [< $VN_L$, Op, $VN_R$ >]
    HT [T] = $VN_T$
def getNumber (operand) :
    if HT[operand] : // if entry in HT alr exists for this value
        return HT[operand]
    else :
        HT[operand] = num
        num += 1
        return HT[operand]
```

find the number that < VN_L, Op, VN_R > is assoc. with, aka HT[< VN_L, Op, VN_R >] Then, find the other entry in HT that has the same number. the key for this entry - aka the other name sharing that num- is returned

Example walkthrough?

$a \leftarrow b + c$

$b \leftarrow a - d$

$c \leftarrow b + c$

~~$d \leftarrow a - d$~~ → $d \leftarrow b$

HT:

num	name
~~0~~ 4	b
~~1~~ 5	c
2	0+1
2	a
~~3~~ 4	d
4	2-3
5	4+1

1. $a \leftarrow b + c$
   - create new entries for b & c → HT[b] = 0 and HT[c] = 1
   - create new entry for "0+1" → HT[0+1] = 2
   - create new entry for a, same num as "0+1" → HT[a] = 2
2. $b \leftarrow a - d$
   - create new entry for d → HT[d] = 3
   - create new entry for "2-3" → HT[2-3] = 4
   - set HT[b] = HT[2-3] = 4
3. $c \leftarrow b + c$
   - create new entry for "4+1" → HT[4+1] = 5
   - set HT[c] = HT[4+1] = 5
4. $d \leftarrow a - d$
   - entry already exists for "2-3"!
   - x = getName [2-3] = entry where num is 4 = b
   - $VN_x = 4$
   - getNum(x) = 4
   - rewrite instruction to: $d \leftarrow b$
   - set HT[d] = HT[2-3] = 4

How can the LVN algorithm be extended for more optimizations?

→ Commutative operations

→ Algebraic identities

→ Constant Folding

How would the LVN use constant folding?

→ Constant Folding: Store values for constants, rather than names.

- For instructions (that are operations) where all operands are constants, LVN can evaluate/perform the operation, & then replace the instruction with a load imm. inst. that loads in the evaluated result.
- Ex: $\left.\begin{array}{l} a \leftarrow 3 \\ b \leftarrow a + 2 \end{array}\right]$ → Rewrite L2 using constant folding → $\left.\begin{array}{l} a \leftarrow 3 \\ b \leftarrow 5 \end{array}\right]$

How would the LVN use commutative operations?

→ Commutative operations: ensure that operations that are communatative, eg $a \times b$ and $b \times a$, receive the same valve numbers/ are treated as the same value

How would the LVN use algebraic identities?

→ Algebraic identities: Keep a dictionary AID of algebraic identities that the LVN can apply to simplify the code, e.g.:

Operator	Expr	Simplified identity
add	$a + 0$	$= a$
sub	$a - 0$	$= a$
sub	$a - a$	$= 0$
mul	$2 \times a$	$= a + a$
mul	$1 \times a$	$= a$
mul	$0 \times a$	$= 0$
div	$a \div 1$	$= a$
div	$a \div a$	$= 1$

Operator	Expr	Simplified identity
exp	a^1	$= a$
exp	a^2	$= a \times a$
srl	$a >> 0$	$= a$
sll	$a << 0$	$= a$
AND	$a \& a$	$= a$
OR	$a \parallel a$	$= a$
MAX	$\max(a, a)$	$= a$
min	$\min(a, a)$	$= a$

- here, each "simplified identity" represents an operation which is equivalent to its corresponding Expr, BUT is simpler/faster/more efficient. For ex, `sub $b $a $a` is equivalent to but less efficient than doing `li $b 0x0`

→ For each instruction $T_i \leftarrow L_i \text{ Op } R_i$, iterate thru all the entries in AID where "operator" is Op. If any of the Exprs match the instruction, replace the instruction with the corresponding simplified version.

→ Performing LVN over an IR in SSA prevents lost values

– Static Single Assignment (SSA) –

What is SSA form?

→ Linear IR + Namespace; encodes control flow & value flow into names

→ Each name corresponds to one definition point in the code

- each defn. has a distinct name
- each use refers to a single defn.

How do we translate a Linear IR into SSA form?

→ At each definition, generate a new name – e.g., by adding a subscript:

$$\begin{array}{l} a \leftarrow x + y \\ b \leftarrow x + y \\ a \leftarrow z \end{array} \longrightarrow \begin{array}{l} a_0 \leftarrow x_0 + y_0 \\ b_0 \leftarrow x_0 + y_0 \\ a_1 \leftarrow z_0 \end{array}$$

→ At each control-flow join, add a Ø-function – a "fake" instruction that lives at the top of a basic block that has multiple predecessors

- the Ø-function basically allows compiler to choose which previous definition's value should get assigned as the value of a new defn., when that value is dependent on the control flow that led up to it.

→ Compiler inserts Ø-functions at points where different control-flow paths merge, and it then renames variables to make the single-assignment property hold.

How do the ϕ-functions work?

→ At control-flow merge points, a var may come from differing previous definitions depending on which path was taken

→ The ϕ operation selects which prev. defn will be used.

small example?

→ Ex: $\boxed{a_2 \leftarrow \phi(x_1, x_2)}$

- When execution enters this block, if we came from predecessor 1, then a_1 shld take the value of x_1. If we came from predecessor 2, then a_2 should take the value of x_2.

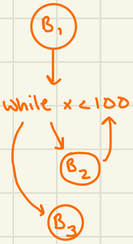
→ SSA attaches each argument of the ϕ to a particular incoming edge in the control-flow graph (CFG); the ϕ-function selects the arg. corresponding to the CFG edge just taken

How would we turn a while block into SSA form?

→ Example linear IR:

```
x ← ...          ]
y ← ...          ]─ B1
while (x<100)
    x ← x+1      ]
    y ← y+x      ]─ B2
... (B3)
```

→ CFG:



→ There are 2 merge points:

1) Top of the loop body, where control could either enter from above (e.g. the first iteration), OR from the bottom of the loop, when starting next iter.

2) The block AFTER the loop, where the val of x could either come from the while loop, or the prev. block if we never entered the loop

→ SSA encoding:

```
    $x_0 \leftarrow ...$
    $y_0 \leftarrow ...$
    bge $x_0$ 100 continue
Loop:
    $x_1 \leftarrow \phi(x_0, x_2)$
    $y_1 \leftarrow \phi(y_0, y_2)$
    $x_2 \leftarrow x_1 + 1$
    $y_2 \leftarrow y_1 + x_2$
Continue:
    $x_3 \leftarrow \phi(x_0, x_2)$
    $y_e \leftarrow \phi(y_0, y_2)$
```

— Superlocal Valve Numbering —

What is Superlocal Valve Numbering (SLVN)?

→ An extension of LVN, applied to a regional scope

→ The regional scope = Extended Basic blocks (EBBs)

What is an EBB?

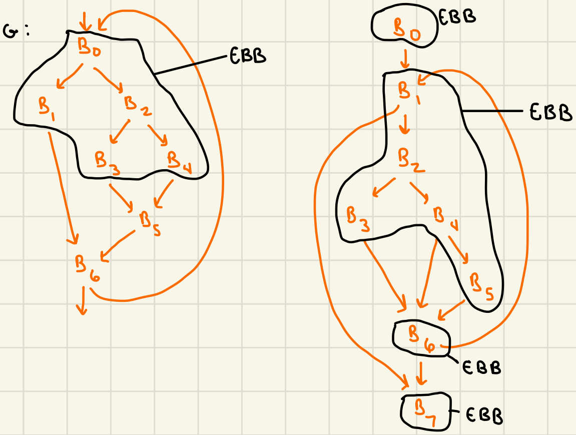
→ A set of basic blocks $\{B_0, B_1, \dots B_n\}$ where:

- B_0 = the entry node, OR B_0 has multiple CFG predecessors, AND
- all other B_i have only one predecessor, which is in the set

→ If one instruction executes, every prior inst. on the CFG path in the EBB also executed

Examples of EBBs?

→ CFG:



How does SLVN work?

→ Apply LVN to paths through EBBs: treat each path thru an EBB as straight-line code & find redundancies that cross basic block boundaries

→ Challenge: eliminating redundant LVN work

Example?

B_0: $m_0 = a_0 + b_0$
$n_0 = a_0 + b_0$
$(a_0 > b_0) \longrightarrow B_1, B_2$

B_1: $p_0 = c_0 + d_0$
$r_0 = c_0 + d_0$
$\longrightarrow B_6$

B_2: $q_0 = a_0 + b_0$
$r_1 = c_0 + d_0$
$(a_0 > b_0) \longrightarrow B_3, B_4$

B_3: $e_0 = b_0 + 18$
$s_0 = a_0 + b_0$
$u_0 = e_0 + f_0$
$\longrightarrow B_5$

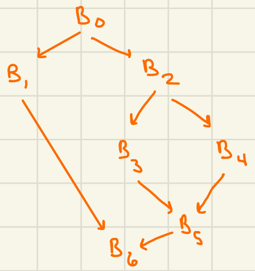
B_4: $e_1 = a_0 + 17$
$t_0 = c_0 + d_0$
$u_1 = e_1 + f_0$
$\longrightarrow B_5$

B_5: $e_2 = \phi(e_0, e_1)$
$u_2 = \phi(u_0, u_1)$
$v_0 = a_0 + b_0$
$w_0 = c_0 + d_0$
$x_0 = e_2 + f_0$
$\longrightarrow B_6$

B_6: $r_2 = \phi(r_0, r_1)$
$y_0 = a_0 + b_0$
$z_0 = c_0 + d_0$

What are the EBBs in this example?

→ The CFG for this code:



→ EBBs:

$\{B_0, B_1, B_2, B_3, B_4\}$ *

$\{B_5\}$

$\{B_6\}$

How would basic SLVN work on the starred EBB?

→ By treating each path as if it were a single block, & performing LVN on each possible path.

→ Possible paths: (B_0, B_1), (B_0, B_2, B_3), (B_0, B_2, B_4)

→ For each path, it would perform LVN on each block in the path, & then simply carry the hash table forward to the next block in the path

How would the basic SVN alg work on this entire block?

1. Create scope (e.g., HT) for B_0
2. Apply LVN to B_0 - eliminates redundant n_0 assignment
3. Create scope for B_1: e.g., carrying forward B_0's HT; apply LVN to B_1
4. Add B_6 to WorkList
5. Delete B_1's scope: e.g., revert changes made to o.g. HT from B_0 when we analyzed B_1
6. Create scope for B_2; apply LVN to B_2
7. Create scope for & apply LVN to B_3
8. Add B_5 to WorkList
9. Delete B_3's scope
10. Create scope for B_4; apply LVN to B_4
11. Delete B_4's scope
12. Delete B_2's scope
13. Delete B_0's scope
14. Create scope for B_5; apply LVN to B_5
15. Delete B_5's scope
16. Create scope for B_6; apply LVN to B_6
17. Delete B_6's scope

What are the drawbacks of this naive approach?

→ Analyzing basic blocks once per path = lots of repeated work; inefficient

What is a better approach?

→ Basic blocks are analyzed once, and ValueNumber table is reused

- checkpoint state at BB boundaries to restore table's state
- unwind effects on VN table by walking BB backward, OR
- use linked list of VN tables & delete leaf tables to start a new path

→ Requires SSA form

Data Flow Analysis

What is Data-flow analysis? (DFA)

→ A way of reasoning (at compile time) about the flow of valves at runtime

→ Goal: analyze prog. in support of optimizations

→ Process: Iterative DFA

What is the scope?

→ global analysis, or full CFG

What are the considerations?

→ Precision, correctness, & termination

What is iterative Data Flow Analysis?

9:01

→ Iterative fixed-point algorithms where the fixed point exists & is unique

→ For each node of the CFG:

1. evaluate an equation
2. update the set defined by the eqn.
3. go back to step 1 until fixed point is reached

→ BENEFIT: simple & robust

- works directly on any valid CFG
- guaranteed to terminate

Why is precision important to consider?

→ DFA is a static analysis; it may be imprecise, & transformations must be conservative

→ Sources of imprecision:

- control flow
- ambiguous references (analysis must assome any feasible valve is possible)
- reading from an external source (adds non-determinism to program)
- external procedures

Why is control flow a source of imprecision?

→ Analysis assumes all paths through the CFG are feasible

→ Analysis is precise up to symbolic execution

Why are procedure calls a source of imprecision?

→ Analysis must assume a callee will use & modify any accessible variable

– Dominance –

What is dominance?

→ Given a CFG with nodes $\{b_0, \ldots b_n\}$ where b_0 = entry node, node b_i dominates node b_j (written $b_i \gg b_j$) if & only if b_i lies on every path from b_0 to b_j

→ By defn, $b_i \gg b_i$ for all b_i

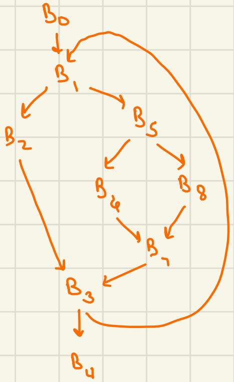
→ Dominance = a tool used by compiler for reasoning about the shape & structure of a CFG

- Important in construction of SSA form

What is a DOM set?

→ Each node b_i in a CFG has a set Dom(b_i), which contains the names of all nodes that dominate b_i

Example of DOM sets given a CFG?



- $DOM(B_0) = \{B_0\}$
- $DOM(B_1) = \{B_0, B_1\}$
- $DOM(B_2) = \{B_0, B_1, B_2\}$
- $DOM(B_3) = \{B_0, B_1, B_3\}$
- $DOM(B_4) = \{B_0, B_1, B_3, B_4\}$
- $DOM(B_5) = \{B_0, B_1, B_5\}$
- $DOM(B_6) = \{B_5, B_0, B_1, B_6\}$
- $DOM(B_7) = \{B_5, B_0, B_1, B_7\}$
- $DOM(B_8) = \{B_5, B_0, B_1, B_8\}$

What is immediate dominance?

→ Given a CFG with nodes $\{b_0, \dots b_n\}$ where b_0 = entry node :

- node b_i is the immediate dominator of b_j iff :
  - → b_i dominates b_j
  - → b_i is the dominating node closest to b_j
  - → $b_i \neq b_j$
- b_0 has no IDOM

Example finding IDOM sets given a CFG?

→ From same CFG as above :

node	IDOM	node	IDOM
0	∅	5	B_1
1	B_0	6	B_5
2	B_1	7	B_5
3	B_1	8	B_5
4	B_3		

How does the compiler compute dominance, conceptually?

→ $$\boxed{DOM(n) = \{n\} \cup \left(\bigcap_{m \in preds(n)} DOM(m) \right)}$$ (and $DOM(b_0) = \{b_0\}$

→ MEANING : For node n, Compute DOM(n) :

1. Add {n} to DOM(n)
2. For each node that precedes n (e.g. every node by which n is reachable), get the Intersection of the DOM sets of each of those nodes. Add this to DOM(n)

How is this computation/algorithm implemented?

What is the proof that this alg considers termination?

→ Iterative fixed-point ALG for a CFG with |N| nodes:

1. initialize $DOM(b_0) = \{b_0\}$
2. For each node $b_1, ... b_n$, initialize $DOM(n) = \{b_0, b_1, ... b_n\}$

   (aka, initialize each node's DOM set as the set of all nodes)
3. For $i=1$ to $|N|-1$:

   $$DOM(b_i) = \{b_i\} \cup \left(\bigcap_{b_m \in preds(b_i)} DOM(b_m) \right)$$
4. Repeat step 3 until, for all i, $DOM(b_i)$ does not change at all (aka fixed point reached)

DFA to accept "not" and "new"

let X = error state

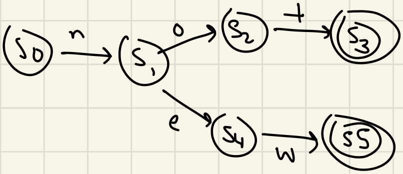
yay compass review!

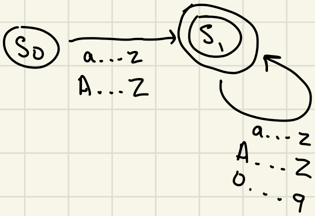
S: {S0, S1, S2, S3, S4, S5}

Σ : {n, o, t, e, w}

δ:

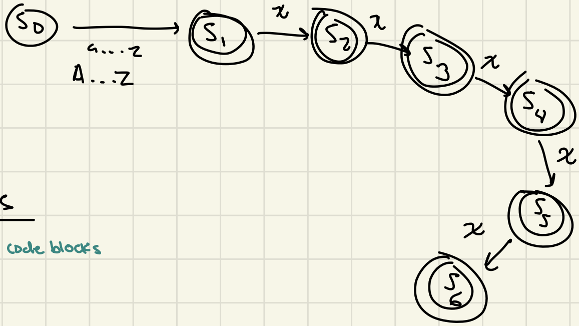
	n	o	t	e	w
S0	S1	x	x	x	x
S1	x	S2	x	S4	x
S2	x	x	S3	x	x
S3	x	x	x	x	x
S4	x	x	x	x	S5
S5	x	x	x	x	x





Identifier with one alphabetic char followed 0 or more alphanumeric chars

let set x = a...z
A...z
0-9



Identifier with one alphabetic char followed by up to 5 alphanumeric chars

COLORS

- Main secondary + code blocks
- Main secondary
- Main purple
- Secondary purple
- neutral purple