Compiler Design Summary

David Hofer

07.01.2022

Contents

TODO

- finish Intermediate Representations
- Lexing: Computing minimal equivalent DFA (Myhill & Nerode algorithm) → do we have to know this?
- look at demos from the lectures
- Parsing: How to determine if a grammar is LL(k) for some k?
- Parsing: Converting the table to code: what are the auxiliary nonterminals?
- Parsing: how to extend LL(1) to LL(k)?
- Find "why"s in summary and answer them
- Parsing: Why does LALR introduce new reduce/reduce conflicts but not shift/reduce?
- End of lec13 and beginning of 14, Operational Semantics/lambda calculus
- SSA & phi nodes

1 General

1.1 Intermediate Representations

For simple languages, lowering from AST to assembly can be done directly, no need for intermediate representations. Idea: Maintain Invariants, e.g. code emitted for a given expression comptues the answer into rax.

Key challenges: storing intermediate values needed to compute complex expressions; some instructions use specific registers (e.g. shift)
One simple strategy: compilation emits instructions into an instruction stream. To compile an expression `"e1 op e2"`, recursively compile its sub-expressions, and process the results.

Invariants:

- Compilation of an expression yields its result in `rax`
- Argument x_i is stored in a dedicated operand
- Intermediate values are pushed onto the stack
- Stack slot is popped after use (so the space is reclaimed)

Resulting code is wrapped to comply with cdecl calling conventions.
$\Longrightarrow$ simple *syntax-directed* translation. Input syntax uniquely determines the output, no complex analysis or code transformation is done. This works fine for simple languages.

But this has many drawbacks.

Intermediate Representations
Abstract machine code, allows machine independent code generation and optimization.
In practice, multiple IR's might be used.
Should be an easy translation target from level above, easy to translate to the level below, and have a narrow interface.

`To be continued`

1.2 LLVM

Check Lecture 7 & 8 slides.

2 Lexing

Lexical Analysis: takes source code as input and turns it into a token stream that will be processed by the parser.

Change the *character stream* `if (b == 0) a = 1;` into *tokens*:
`IF; LPAREN; Ident("b"); EQEQ; Int(0); RPAREN; LBRACE; Ident("a"); EQ; Int(1); SEMI; RBRACE;`
Token: data type that represents indivisible "chunks" of text, e.g.

- Identifiers
- keywords
- integers
- floating point

- symbols
- strings
- comments

2.1 Regular Expressions

regular expression R =

- 'a': an ordinary character stands for itself
- ε: empty string
- R_1 | R_2: choice of R_1 or R_2
- R_1R_2: concatenation
- R*: zero or more repetitions of R
- R+: RR*
- R?: R | ε
- ['a' – 'z']: (a | b | ... | z)
- [^'0' – '9']: any character except 0 through 9
- R as x: name the string matched by R as x

Examples:

```
let lowercase = ['a'-'z']
let uppercase = ['A'-'Z']
let character = uppercase | lowercase
```

What if there are multiple possible matches? Most languages choose "longest match".
Conflicts arise for tokens whose regular expressions have a shared prefix.
e.g.

```
let IF = 'if'
let Ident = (lowercase | uppercase | number | '_')*
```

How to resolve? $\Longrightarrow$ ties broken by giving some matches higher priority
usually specified by the order the rules appear in in the lex input file

2.2 DFA/NFA

A deterministic finite automaton consists of a finite set of states and transitions between these states associated with an input symbol (that gets "consumed" when going over that transition) or ε.
Additionally, there is a single starting state and a set of accepting states.

Nondeterministic finite automatons can additionally have two transitions leaving the same state that have the same label.

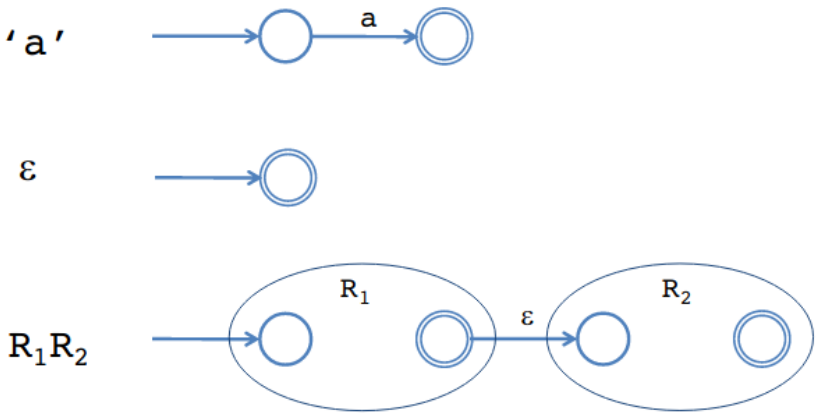
2.2.1 DFA vs. NFA

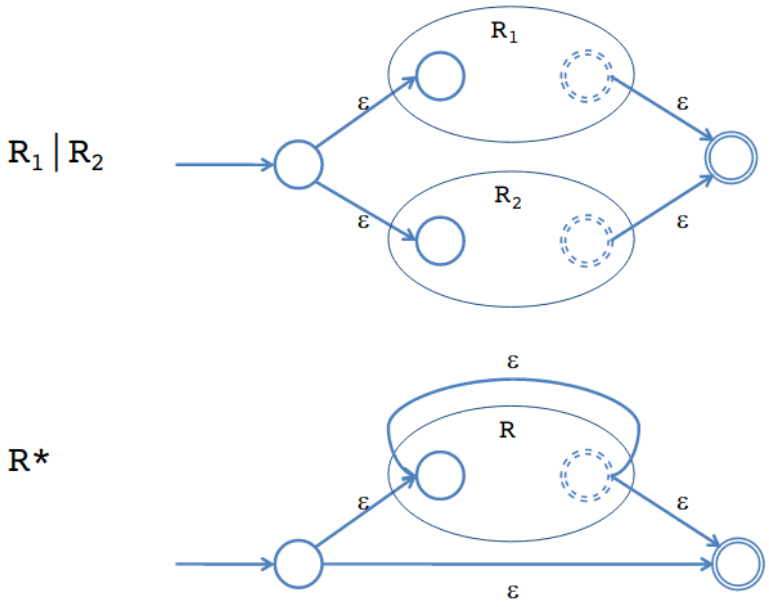
DFA:
- action for each input fully determined
- accepts if input is completely consumed upon reaching an accepting state
- obvious table-based implementation

NFA:
- automaton potentially has a choice at every step
- accepts an input if there *exists* a way to reach an accepting state
- less obvious how to implement efficiently

We can represent every regular expression as a finite automaton. How to convert from RE to NFA?

Assume each NFA has one start state, unique accepting state.





2.2.2 NFA to DFA conversion

1. Let Q be the set of states of the NFA, and Q' the new set of states of the DFA to be constructed. In he beginning, Q = ∅. Let T and T' be the transition tables of the NFA and DFA.

2. Add start state q0 to Q'
   Add transitions of the start state to T'
   If the start state makes multiple transitions for a single symbol, then treat the resulting set of states as a single new state (e.g. from q0 with 'a' can go to q0 or q1 → new state {q0,q1})

3. If any new state is now present in T':
   add the new state to Q'
   add the transitions of that state to T'. For composed states: $\delta(\{q0,q1\},'a')$ = $\delta(q0,'a') \cup \delta(q1,'a')$

4. Keep repeating step 3 until no new state is present in the transition table T' → finished transition table T' of the DFA

5. The accepting states of the DFA are the states containing an accepting state of the NFA

2.3 Lexer Generator

Reads a list of regular expressions, one per token.
Each token has an attached action, which is just a piece of code to run when the regular expression is matched.

```
rule token = parse
```

```
| '-'?digit+ { Int (Int32.of_string (lexeme lexbuf)) }
| '+' { PLUS }
| 'if' { IF }
| character (digit|character|'_')* { Ident (lexeme lexbuf) }
| whitespace+ { token lexbuf }
```

Generates scanning code that decides whether the input is of the form `(`R_1 `|` ... `|` R_n`)*`
After matching a (longest) token, runs the associated action.

2.3.1 Complete Lexer Generator Behavior

- Take each regulare expression R_i and its action A_i
- Compute the NFA formed by (R_1 | ... | R_n); remember the actions associated with the accepting states of the R_i
- NFA to DFA conversion
  Note: there may be multiple accept states (why?); a single accept state may correspond to one or more actions (why?)
- Compute the minimal equivalent DFA; there is a standard algorithm due to Myhill & Nerode
- Produce the transition table
- Implement the longest match:
  - start from initial state
  - follow transitions, remember last accept state entered (if any)
  - perform the highest-priority action associated with the last accept state; if no accept state there is a lexing error

3 Parsing

Find the syntactic structure of the code. Given Token Stream, create an Abstract Syntax Tree.
Traverse the token stream and build a tree representing the "abstract" syntax during that process.
Finite Automatons are not expressive enough for various language concepts, so we'll need something better → Context-Free Grammars (next level in Chomsky Hierarchy.

3.0.1 CFGs mathematically

Example: The balanced parentheses language
`S` → `(S)S`
`S` → ε

A context-free grammar consists of
- a set of *terminals* (e.g. a lexical token)
- a set of *nonterminals* (e.g. S and other syntactic variables)

- a designated nonterminal called the *start symbol*
- a set of *productions*: `LHS` → `RHS`
– LHS is a nonterminal
– RHS is a *string* of terminals and nonterminals

Careful: every nonterminal should eventually rewrite to an alternative that contains only terminal symbols, otherwise the grammar is empty!

3.0.2 Derivation Orders

There are often multiple different derivations possible for a given string. Two standard orders:
- Leftmost derivation
- Rightmost derivation

Both strategies yield the same parse tree!

3.0.3 Associativity

`S` → `E + S | E`
`E` → `number | (S)`

This grammar makes '+' right associative and right recursive.

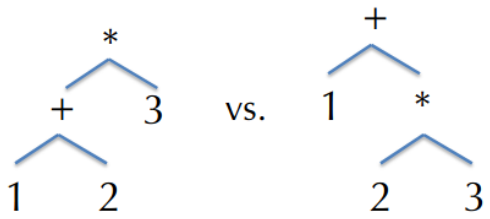
3.0.4 Ambiguity

`S` → `S + S | (S) | number`

This grammar accepts the same set of strings as the previous one. The difference is that this one allows two different derivations for the same string! It is ambiguous.
Moreover, if there are multiple operations, ambiguity in the grammar leads to ambiguity in their *precedence*.
Example:
`S` → `S + S | S * S | (S) | number`

Ambiguous parses/two parse trees:

Input: 1 + 2 * 3
- One parse = (1 + 2) * 3 = 9
- The other = 1 + (2 * 3) = 7



We can often **eliminate ambiguity** by adding nonterminals and allowing recursion only on the left (or right). Higher precedence operators go farther from the start symbol. For the previous grammar, want to
- make '*' higher precedence than '+'
- make '+' left associative
- make '*' right associative

$\Longrightarrow$
$S_0 \rightarrow S_0 + S_1$ `|` S_1
$S_1 \rightarrow S_2 * S_1$ `|` S_2
$S_2 \rightarrow$ `number` `|` `(`S_0`)`

left associative <=> left recursive
right associative <=> right recursive

3.0.5 CFG Summary

An unambiguous CFG specifies how to parse, i.e. convert a token stream to a parse tree. Ambiguity can (often) be removed by encoding precedence and associativity in the grammar. Even with an unambiguous CFG, there may be more than one derivation (but all derivations correspond to the same abstract syntax tree). Still to come: *how to find a derivation*?

3.1 Top-Down/Bottom-Up Parsing

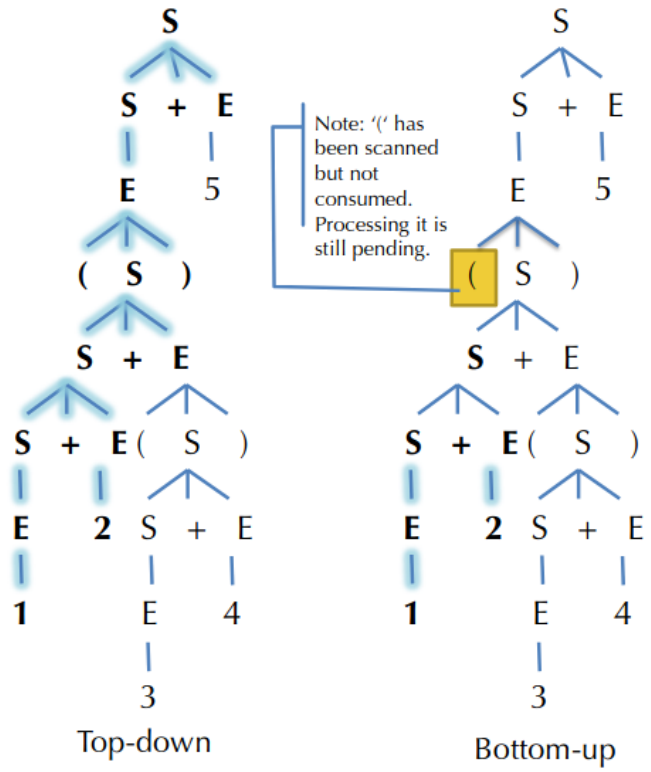
Consider the left-recursive grammar

```
S → S + E | E
E → number | (S)
```

Input string: `(1 + 2 + (3 + 4)) + 5`

What part of the tree must we know after scanning just $\boxed{(1 + 2}$*?*

In Top-down, must be able to guess wich productions to use.



Problem: just given one lookahead symbol, it can be unclear which production to apply.

$\Longrightarrow$ **not all grammars can be parsed top-down with a single lookahead**

3.2 LL

3.2.1 LL(1) grammar

LL(1) means
- Left-to-right scanning
- Left-most derivation
- 1 lookahead symbol

The grammar seen previously isn't "LL(1)"
Problem: we can't decide which S production to apply until we see the symbol after the first expression.

3.2.2 Making a grammar LL(1)

1. Left-factor the grammar. There is a common prefix for each production choice of terminal S, so add a new non-terminal S' at the decision point

```
S → E + S | E
E → number | (S)
```

$\Longrightarrow$

S $\rightarrow$ ES'
S' $\rightarrow \varepsilon$
S' $\rightarrow$ + S
E $\rightarrow$ number | (S)

First production rule turned into 3 new ones, eliminating the common prefix 'E'.

2. Eliminate left-recursion. To avoid infinite looping.
For every rule
S $\rightarrow$ Sα_1 | ... | Sα_n | β_1 | ... | β_m

rewrite as
S $\rightarrow$ β_1S' | ... | β_mS'
S' $\rightarrow$ α_1S' | ... | α_nS' | ε

Consider:
S $\rightarrow$ S + E | E
E $\rightarrow$ number | (S)

$\Longrightarrow$

S $\rightarrow$ E S'
S' $\rightarrow$ + E S' | ε
E $\rightarrow$ number | (S)

3.2.3 Predictive Parsing

Given an LL(1) grammar, for a given nonterminal, the lookahead symbol uniquely determines the production to apply.
Top-down parsing = predictive parsing
Driven by a predictive parsing table: nonterminal x input token $\rightarrow$ production.
Note: it is convenient to add a special *end-of-file* token $ and a start symbol T (top-level) that requires $.
Final LL(1) grammar:
T $\rightarrow$ S$
S $\rightarrow$ ES'
S' $\rightarrow \varepsilon$
S' $\rightarrow$ + S
E $\rightarrow$ number | (S)

	number	+	(	)	$ (EOF)
T	↦ S$		↦S$		
S	↦ E S′		↦E S′		
S′		↦ + S		↦ ε	↦ ε
E	↦ number		↦ (S)		

3.2.4 How do we construct the parse table?

Consider a given production `A` $\rightarrow \gamma$

Case 1: Construct the set of all input tokens that may appear *first* in strings that can be derived from γ. Add the production $\rightarrow \gamma$ to the entry `(A,token)` in the parse table for each such token.

Case 2: If γ can derive ε (empty string), then we construct the set of all input tokens that may *follow* the nonterminal `A` in the grammar. Add the production $\rightarrow \gamma$ to the entry `(A,token)` in the parse table for each such token.

IMPORTANT: if there are two different productions for a given entry, the grammar is not LL(1).

3.2.5 Converting the table to code

Define N mutually recursive functions, one for each nonterminal A: `parse_A`.

make `parse_A` be of type `unit -> ast` if A is not an auxiliary nonterminal. Otherwise, `parse_A` takes extra ast's as inputs, one for each nonterminal in the "factored" prefix.

Given the parse table, each function `parse_A`
- "Peeks" at the lookahead token
- follows the production rule in the corresponding entry
- consumes terminal tokens from the input streams
- calls `parse_X` to create a sub-tree for nonterminal X
- if the rule ends in an auxiliary nonterminal, call it with appropriate ast's (the auxiliary rule creates the ast after looking at more input)
- otherwise, this function builds the ast tree itself and returns it

TODO: what are these auxiliary nonterminals?

3.2.6 LL(1) Summary

- top-down parsing that finds the leftmost derivation
- Language grammar $\Longrightarrow$
  LL(1) grammar $\Longrightarrow$
  prediction table $\Longrightarrow$
  recursive-descent parser

- Problems:
  - Grammar must be LL(1)
  - Can extend to LL(k) (it just makes the table bigger) TODO: how to?
  - Grammar cannot be left recursive (parser functions will loop!)

3.3 LR

LR(k) parser - **L**eft-to-right scanning
- **R**ight-most derivation
- **k** lookahead symbols

LR grammars are more expresive than LL!
Can handle left-recursive and right recursive grammars → virtually all programming languages. Easier to express programming language syntax (e.g. no left factoring)

Technique: Shift-Reduce parsers
- Work bottom up instead of top down
- Construct right-most derivation of a program in the grammar
- used by many parser generators (yacc, menhir, etc)
- better error detection/recovery
- poor error reporting (GCC's shift from bottom-up to top-down parsing)

3.3.1 Shift/Reduce Parsing

Parser state:
- Stack of terminals and nonterminals
- Unconsumed input is a string of terminals
- Current derivation step is `stack + input`

Parsing is a sequence of *shift* and *reduce* operations.
Shift: Move look-ahead token to the stack
Reduce: Replace symbols γ at top of stack with nonterminal X s.t. $X \to \gamma$ is a production, i.e. `pop` γ`, push X`

Stack	Input	Action
	(1 + 2 + (3 + 4)) + 5	shift (
(	1 + 2 + (3 + 4)) + 5	shift 1
(1	+ 2 + (3 + 4)) + 5	reduce: E ⟼ number
(E	+ 2 + (3 + 4)) + 5	reduce: S ⟼ E
(S	+ 2 + (3 + 4)) + 5	shift +
(S +	2 + (3 + 4)) + 5	shift 2
(S + 2	+ (3 + 4)) + 5	reduce: E ⟼ number

3.3.2 Action Selection Problem

Given a stack σ and a look-ahead symbol **b**, should the parser
- shift **b** onto the stack (new stack is σ**b**), or
- reduce a production `X` $\to \gamma$, assuming that $\sigma = \alpha\gamma$ (new stack is αX)?

Sometimes parser can reduce, but should not (e.g. `X` $\to \varepsilon$ can always be reduced)
Sometimes the stack can be reduced in different ways (why?)

Main idea: decide based on a prefix α of the stack plus look-ahead
The prefix α is different for different possible reductions since in productions `X` $\to \gamma$ and `Y` $\to \beta$, γ and β might have different lengths

Goal: want to know what set of reductions are legal at any point.
How do we keep track?

3.3.3 LR(0) parsing

General goal: Want to know what set of reductions are legal at any given point.
Do this by summarizing all possible stack prefixes α as a finite parser state.
Parser state is computed by a DFA that reads the stack σ.
Accept states of the DFA correspond to unique reductions to apply.

Do this by LR(0) parsing.
It's too weak to handle many language grammars (e.g. the sum grammar), but helpful for understanding how shift-reduce parsers work.

$\to$ parser state, shift, reduce as described above.

3.3.4 LR(0) states

LR(0) state: (set of) items to track progress on possible upcoming reductions
LR(0) item: a production with an extra separator "." in the RHS

$$S \mapsto (L) \mid id$$
$$L \mapsto S \mid L , S$$

Example items: $S \mapsto .(L)$ or $S \mapsto (. L)$ or $L \mapsto S.$

Intuition:
Stuff before the '.' already on stack (beginnings of possible γ's to be reduced)
Stuff after the '.' is what might be seen next
The prefixes α are represented by the state itself
$\to$ use a DFA to recognize this LL(0) grammar

3.3.5 Constructing the LR(0) DFA

We will use a DFA to parse LR(0) grammars.

Example grammar:
`S` $\to$ `(L) | id`
`L` $\to$ `S | L, S`

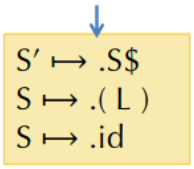
First step: Add a new production `S' → S$` to the grammar.
`S' → S$`
`S → (L) | id`
`L → S | L, S`

Start state of the DFA = empty stack, so it contains the item
`S' → .S$`

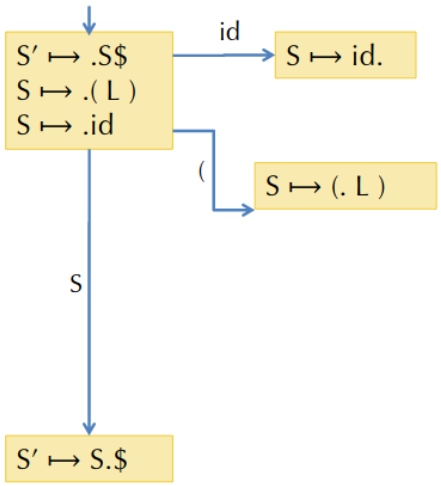
Closure of a state
- Adds items for all productions whose LHS nonterminal occurs in an item in the state just after the '.' (e.g. `S` in `S' → .S$`)
- The added items have the '.' located at the beginning (no symbols for those items have been added to the stack yet)
- Note that newly added items may cause yet more items to be added to the state (if the '.' is again located in front of a nonterminal)... keep iterating until a *fixed point* is reached

Example:
`CLOSURE({S' → .S$}) = {S' → .S$, S → .(L), S → .id }`

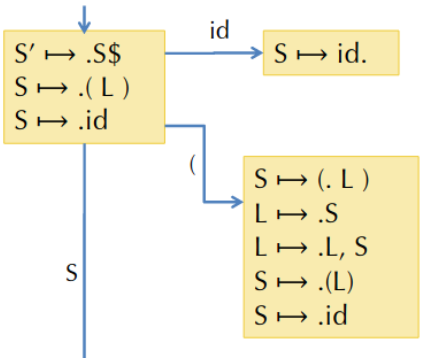
Resulting "closed state" contains the set of all possible productions that might be reduced next, and represents a state of the DFA.



Next, we add the transitions.
For each symbol that can occur after the '.' in any of our items, add a transition to a new state. The edge label of this transition is the symbol.
The target state initially contains all items from the source state that have the edge-label symbol after the '.', but we advance the '.' (to simulate shifting the item onto the stack).



For each new state, we take the closure (in as many iterations as it takes to reach a fixed point for this closure)

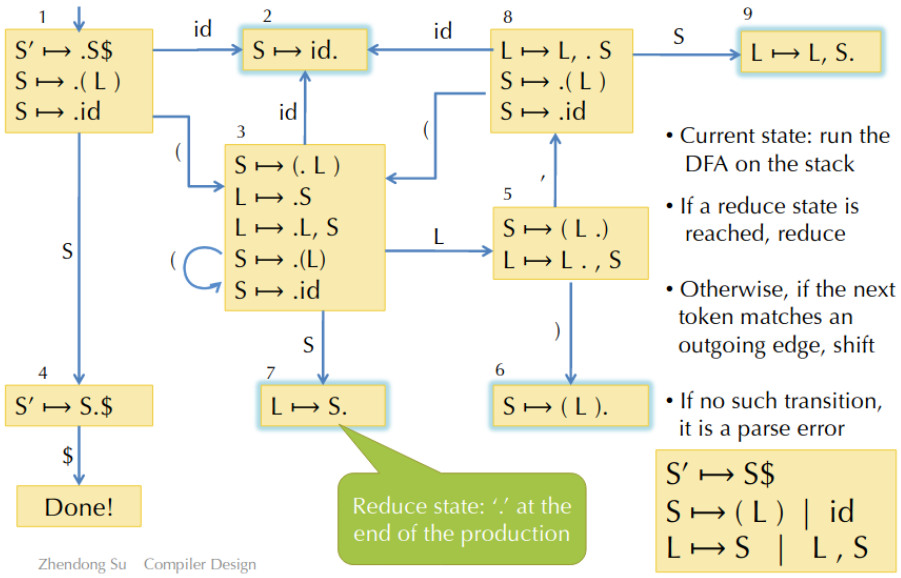


Then, we compute the new transitions for the new states.

Important: this only works, if in any reduce state there is only 1 possible reduce action!

Full procedure

- Construct a state with the initial item `S'` → `.S$`
- Compute the closure of that state; this is the initial state of the DFA
- Add transitions for all possible symbols after the '.'
- For each new state, compute the closure and add transitions
- Repeat.



The final, complete DFA for our example grammar.

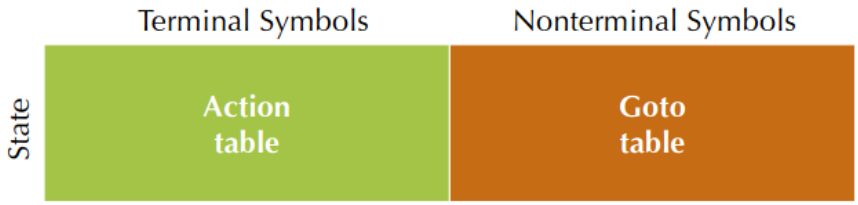
3.3.6 Running the DFA

Run parser stack through the DFA.
If resulting state is not a reduce state:
shift the next input symbol and transition with respect to the DFA
Else:
pop RHS from stack and push LHS

Optimization: no need to rerun DFA from beginning each step (after reduce)
- Store the state with respect to each symbol on the stack: e.g. ${}_1(_3(_3L_5)_6$
- On a reduction `X` $\rightarrow \gamma$, pop stack to reveal the state too: e.g. reduce `S` $\rightarrow$ `(L)` to reach stack ${}_1(_3$
- Push the reduction symbol: e.g. to reach stack ${}_1(_3S$
- Continue with parsing from the revealed state: ${}_1(_3S_7$

3.3.7 Implementing the parsing table

Represent the DFA as a table of shape `state * (terminals + nonterminals)`
Entries for the "action table" specify two kinds of actions:
- Shift and go to state n
- Reduce using reduction `X` $\rightarrow \gamma$: pop γ off the stack to reveal the state, then look up X in the "goto table" and go to that state



Example parse table:

	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S↦id	S↦id	S↦id	S↦id	S↦id		
3	s3		s2			g7	g5
4					DONE		
5		s6		s8			
6	S ↦ (L)	S ↦ (L)	S ↦ (L)	S ↦ (L)	S ↦ (L)		
7	L ↦ S	L ↦ S	L ↦ S	L ↦ S	L ↦ S		
8	s3		s2			g9	
9	L ↦ L,S	L ↦ L,S	L ↦ L,S	L ↦ L,S	L ↦ L,S		

sx = shift and go to state x
gx = go to state x

3.3.8 LR(0) limitations

An LR(0) machine only works if states with reduce actions have a *single* reduce action. In such states, the machine always reduces, ignoring lookahead.

With more complex grammars, the DFA construction will yield states with **shift/reduce** and **reduce/reduce** conflicts.
OK: single reduce or multiple shift items in one state
shift/reduce: reduce as well as shift item in one state `S → (L).; L → .L, S`
reduce/reduce: two reduce items in one state `S → L, S.; S → , S.`

Example: right associative "sum" grammar
`S → E + S | E`
`E → number | (S)`

This grammar is not LL(0). The DFA will run into a shift/reduce conflict: from starting state, following transition 'E', we could now reduce `S → E.` or shift `S → E. + S`

Ambiguities in associativity/precedence often lead to shift/reduce conflicts.

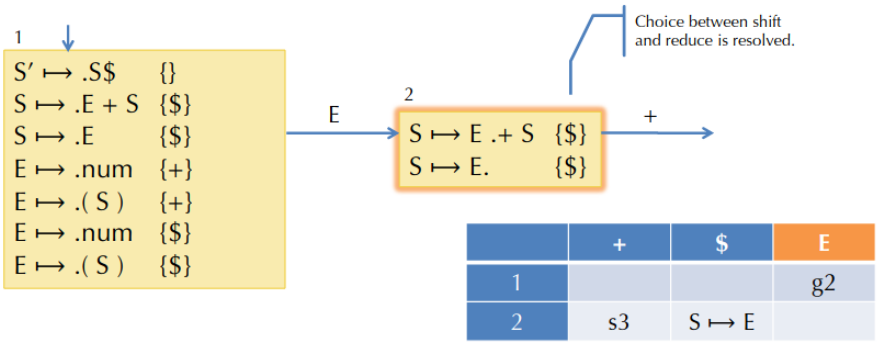
LR(1) to the rescue: such conflicts can often be resolved using a single look-ahead.

3.3.9 LR(1) parsing

LR(1) state = set of LR(1) items
LR(1) item = LR(0) item + a set of look-ahead symbols: $\texttt{A} \rightarrow \alpha.\beta \ , \ \mathcal{L}$
LR(1) closure is more complex:
- Form the set of items just as for LR(0) algorithm
- Whenever a new item $\texttt{C} \rightarrow \gamma$ is aded because $\texttt{A} \rightarrow \beta.\texttt{C}\delta \ , \ \mathcal{L}$ is already in the set, we need to compute its look-ahead set $\mathcal{M}$.
1. the look-ahead set $\mathcal{M}$ includes FIRST(δ) (the set of terminals that may start strings derived from δ)
2. If δ is or can derive ε, then the look-ahead $\mathcal{M}$ also contains $\mathcal{L}$

3.3.10 Using the LR(1) DFA

This now resolves the previous conflicts.



Fragment of the Action & Goto tables

In state 2, if lookahead is a '+' we shift, if it is a '$' we reduce.

The behavior is determined if
- There is no overlap among the look-ahead sets for each reduce item, and
- None of the look-ahead symbols appear to the right of a '.'

LR(1) gives maximal power out of a 1 look-ahead symbol parsing table. DFA + stack is a push-down automaton.

In practice, LR(1) tables are big. Modern implementations (e.g. menhir) directly generate code.

3.4 LR variants

3.4.1 LALR

Consider for example the LR(1) states
$\{ \ \texttt{X} \rightarrow \alpha. \ , \ \texttt{a} \ ; \ \texttt{Y} \rightarrow \beta. \ , \ \texttt{c} \}$

{ X $\rightarrow \alpha.$, b ; Y $\rightarrow \beta.$, d}

They have the same core and can be merged.
The merged state contains
{ X $\rightarrow \alpha.$, a/b ; Y $\rightarrow \beta.$, c/d}

These are called LALR(1) states, LookAhead LR.
Typically 10 times fewer LALR(1) states than LR(1).

Compared to LR(1), LALR(1) may introduce new reduce/reduce conflicts, but not new shift/reduce conflicts. Why?

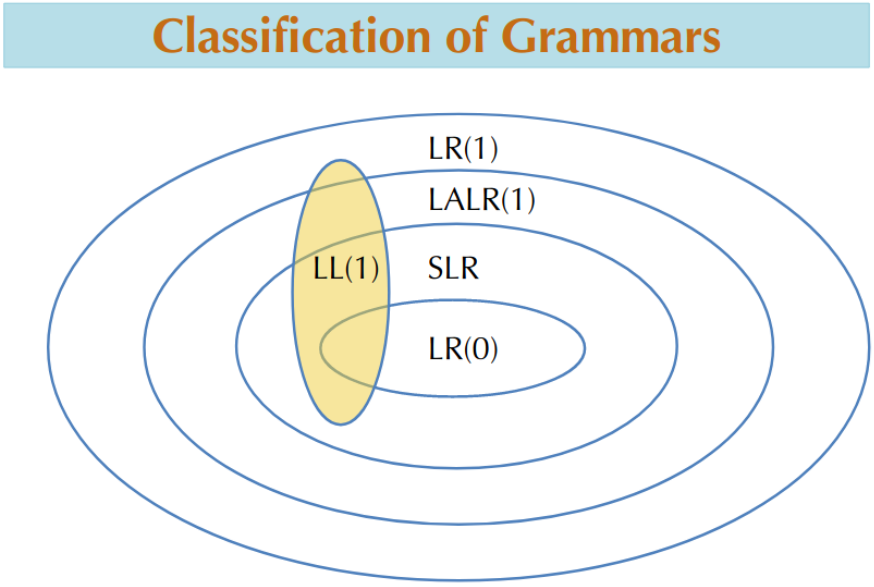
{X $\rightarrow \alpha.$, a/_ ; Y $\rightarrow \beta$.a , _}

3.4.2 SLR(1)

"Simple" LR: like LR(0), but use the FOLLOW information

3.4.3 GLR

"Generalized" LR:
- Efficiently compute the set of *all* parses for a given input
- Later passes should disambiguate based on other context



3.5 If/Else, Dangling Else Problem

Consider this grammar
S $\rightarrow$ if (E) S
S $\rightarrow$ if (E) S else S
S $\rightarrow$ X=E

`E` → `...`

Consider how to parse
`if (E1) if (E2) S1 else S2`

Known as the "Dangling Else" problem.

Want to rule out: `if (E1) { if (E2) S1 } else S2`
Observation: An un-matched 'if' should not appear as the 'then' clause of a containing 'if'

Solution: change grammar

S ↦ M | U	// M = "matched", U = "unmatched"
U ↦ **if** (E) S	// Unmatched 'if'
U ↦ **if** (E) M **else** U	// Nested if is matched
M ↦ **if** (E) M **else** M	// Matched 'if'
M ↦ X = E	// Other statements

Alternative: enforce use of {} for if/else statements, can also introduce 'else if'

4 Inference Rules and Lambda Calculus

4.1 Inference Rules

Types/uses of inference:
Type-Checking tries to prove a judgment `G; L ⊢ e : t`,
by searching backward through the rules
A judgment `G;L⊢e:t` is read "the expression e is well typed and has type t".

Compiling is a set of inference rules specifying `G ⊢ source ⟹ target`
Compiling is "interpreting" the type checking rules ⟦`C ⊢ e : t`⟧ in the target language.
Compilation judgments are similar to the type checking judgments

What about ⟦`C`⟧? Source level `C` has bindings like `x:int, y:bool`. ⟦`C`⟧ maps source identifiers to source types and ⟦x⟧ (it is the context of the variables?).

4.2 Lambda Calculus

minimal programming language, turing complete.
consisting of variables, functions and function application

4.2.1 Values and Substitution

The only values are (closed) functions:
val ::= fun x -> exp *functions are values*

To *substitute* value v for variable x in expression `e`,

- replace all free occurrences of x in e by v
- in ocaml: written `subst v x e`
- in math: written e{v/x}

Function application is interpreted by substitution:
`(fun x -> fun y -> x + y) 1`
= `subst 1 x (fun y -> x + y)`
= `(fun y -> 1 + y)`

Substitution function

x{v/x}	= v	*(replace the free x by v)*
y{v/x}	= y	*(assuming $y \neq x$)*
(fun x -> exp){v/x}	= (fun x -> exp)	*(x is bound in exp)*
(fun y -> exp){v/x}	= (fun y -> exp{v/x})	*(assuming $y \neq x$)*
$(e_1\ e_2)${v/x}	= (e_1{v/x} e_2{v/x})	*(substitute everywhere)*

4.2.2 Free variables and scoping

Variable `x` is *free* in `fun y -> x + y`.
Free variables are defined in an outer scope.

Variable `y` is *bound* by "`fun y`".
Its scope is the body "`x + y`" in `fun y -> x + y`.

A term with no free variables is called *closed*.
A term with one or more free variables is called *open*.

4.2.3 Free variable calculation

fv := "free variables"

fv(`x`) = {x}
fv(`fun x -> exp`) = fv(exp) \{x} *'x' is bound in exp*
fv(`exp1 exp2`) = fv(exp1) ∪ fv(exp2)

4.2.4 Variable capture and alpha equivalence

If we try to naively "substitute" an open term, a bound variable might *capture* the free variables.

But the names of bound variables don't actually matter.
`fun x -> y x` is "the same" as `fun z -> y z`.

Two terms that differ only by consistent renaming of bound variables are called *alpha equivalent*.

The names of free variables however do matter!

Consider the substitution operation `e1{e1/x}`.

As a fix for variable capture, we define substitution to pick an alpha equivalent version of e1 such that the bound names of e1 don't mention the free names of e2, then we do the "naive" substitution.

Example:
`(fun x -> (x y)) {(fun z -> x) / y}`
`(fun x' -> (x' (fun z -> x))` | *rename x to x'*

5 Compiling Stuff

Operational semantics
`exp` ⇓ `v` := program `exp` evaluates to value `v`.

Values evaluate to themselves:
`v` ⇓ `v`

$$\frac{\text{exp}_1 \Downarrow (\text{fun x -> exp}_3) \qquad \text{exp}_2 \Downarrow \text{v} \qquad \text{exp}_3\{\text{v/x}\} \Downarrow \text{w}}{\text{exp}_1\ \text{exp}_2\ \Downarrow \text{w}}$$

Figure 1: Evaluating function application

Contexts

Through our notion of `C` and ⟦`C`⟧ (a map from source identifiers to types and target identifiers), we get

`x:t ∈ C` means that

(1) `lookup ⟦C⟧ x = (t, %id_x)`

(2) the target type of `%id_x` is ⟦`t`⟧`*` (a pointer to ⟦t⟧)

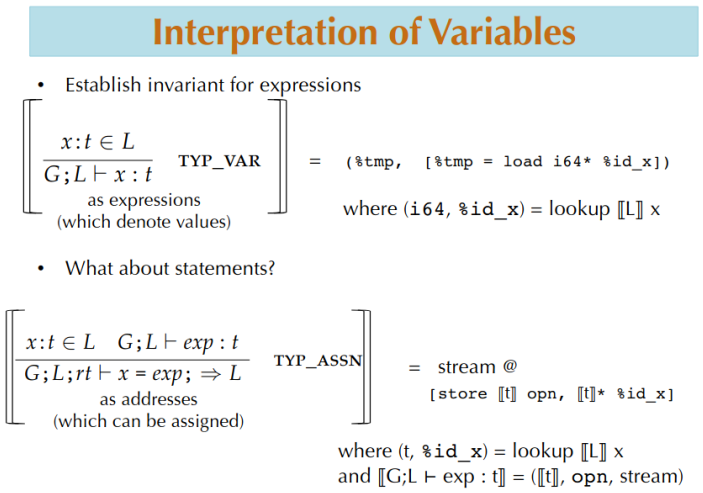


Figure 2: Interpretation of variables

Other judgments:

Statement: ⟦`C; rt ⇒ C'`⟧ `= ⟦C'⟧, stream`

Declaration: ⟦`G;L ⊢ t x = exp ⇒ G;L;x:t`⟧ `= ⟦G;L;x:t⟧,stream`

Here we have the invariant, that stream is of form `stream' @ ( %id_x = alloca ⟦t⟧; store ⟦t⟧ opn, ⟦t⟧* %id_x )` and ⟦`G;L ⊢ exp : t`⟧ `= (⟦t⟧, opn, stream')`

rest follow similarly

5.1 Compiling control

`while(d) s`:
Test conditional `e`, if true jump to body s, else jump to label after body s.

$$[\![C;rt \vdash \texttt{while(e) s} \Rightarrow C']\!] = [\![C']\!],$$

```
lpre:
   opn = [[C ⊢ e : bool]]
   %test = icmp eq i1 opn, 0
   br %test, label %lpost, label %lbody
lbody:
    [[C;rt ⊢ s ⇒ C']]
    br %lpre
lpost:
```

`If-then-else`
Else branch is optional

$$[\![C;rt \vdash \texttt{if (e}_1\texttt{) s}_1 \texttt{ else s}_2 \Rightarrow C']\!] = [\![C']\!],$$

```
   opn = [[C ⊢ e : bool]]
   %test = icmp eq i1 opn, 0
   br %test, label %else, label %then
then:
    [[C;rt ⊢ s1 ⇒ C']]
    br %merge
else:
    [[C;rt ⊢ s2 ⇒ C']]
    br %merge
merge:
```

5.2 Short Circuit Compilation and Evaluation

Usually, we want the translation ⟦e⟧ to produce a value: ⟦`C ⊢ e : t`⟧ = `(ty, operand, stream)`.
But when the expression we're compiling appears in a test, the program jumps to one label or another after the comparison but otherwise never uses the value. In many cases, we want to avoid "materializing" the value (i.e. store in tmp).

⇒ conditional branch translation of Booleans, without materializing value
⟦`C ⊢ e : bool@`⟧ `ltrue lfalse = stream`

⟦C, rt ⊢ if (e) then s1 else s2 ⇒ C′⟧ = ⟦C′⟧,

```
    insns3
then:
    ⟦s1⟧
    br %merge
else:
    ⟦s2⟧
    br %merge
merge:
```

where
⟦C, rt ⊢ s_1 ⇒ C′⟧ = ⟦C′⟧, $insns_1$
⟦C, rt ⊢ s_2 ⇒ C′⟧ = ⟦C′⟧, $insns_2$
⟦C ⊢ e : bool@ ⟧ `then else` = $insns_3$

Short circuit compilation: expressions

- ⟦C ⊢ e : bool@⟧ ltrue lfalse = insns

$$\frac{}{⟦C \vdash \text{false} : \text{bool@}⟧\ \text{ltrue lfalse} = \texttt{[br \%lfalse]}}\ \text{FALSE}$$

$$\frac{}{⟦C \vdash \text{true} : \text{bool@}⟧\ \text{ltrue lfalse} = \texttt{[br \%ltrue]}}\ \text{TRUE}$$

$$\frac{⟦C \vdash e : \text{bool@}⟧\ \text{lfalse ltrue} = \text{insns}}{⟦C \vdash \text{!e} : \text{bool@}⟧\ \text{ltrue lfalse} = \text{insns}}\ \text{NOT}$$

$$\frac{⟦C \vdash e1 : \text{bool@}⟧\ \text{ltrue right} = \text{insns}_1 \quad ⟦C \vdash e2 : \text{bool@}⟧\ \text{ltrue lfalse} = \text{insns}_2}{⟦C \vdash e1 \,|\, e2 : \text{bool@}⟧\ \text{ltrue lfalse} =}$$

```
    insns1
right:
    insn2
```

$$\frac{⟦C \vdash e1 : \text{bool@}⟧\ \text{right lfalse} = \text{insns}_1 \quad ⟦C \vdash e2 : \text{bool@}⟧\ \text{ltrue lfalse} = \text{insns}_2}{⟦C \vdash e1\&e2 : \text{bool@}⟧\ \text{ltrue lfalse} =}$$

```
    insns1
right:
    insn2
```

where `right` is a fresh label

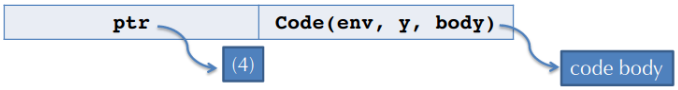
5.3 Variable Scoping

How to determine whether a declared variable is in scope?
→ Compiler keeps a mapping from variables to information about them, using a symbol table.

5.4 Closure Conversion

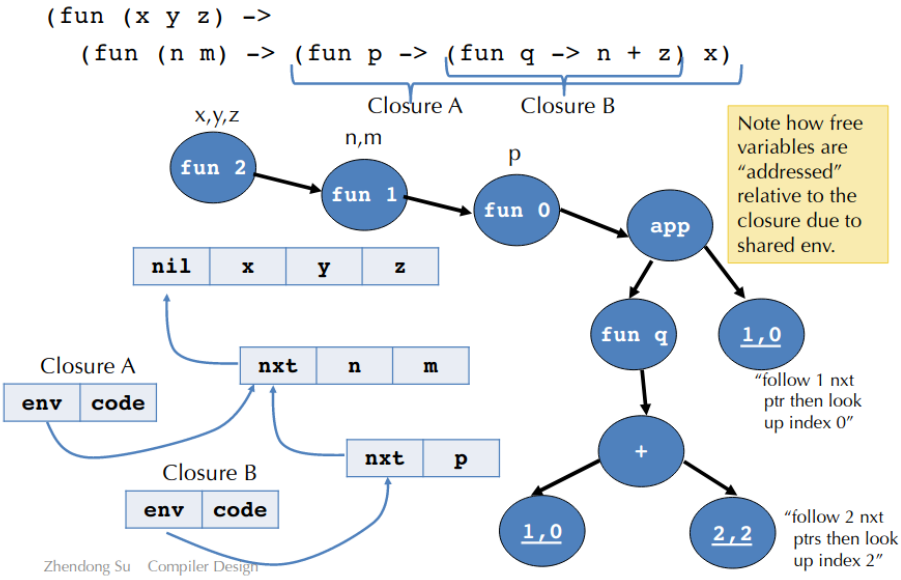
Inner functions can refer to variables bound in outer function.
"Eliminating free variables by packaging up the needed environment in the data structure".
A *closure* is a pair of the environment and a code pointer

Example: `let add = fun x -> fun y -> x + y`
Consider the inner function `fun y -> x + y`
When executing code like `add 4`, the program builds a closure and returns it:
The code pointer takes parameters *env* and *y* plus the code body.



How to implement this/do this exactly, there are many different ways.
E.g. store only the values for free variables in the body of the closure; share sub-components of the environment to avoid copying; use vectors or arrays rather than linked structures

Array-based closures with N-ary functions



5.5 Types

To do type checking, use inference rules to create a derivation or proof tree to do type checking, use inference rules to create a derivation or proof tree.

5.5.1 Type Safety

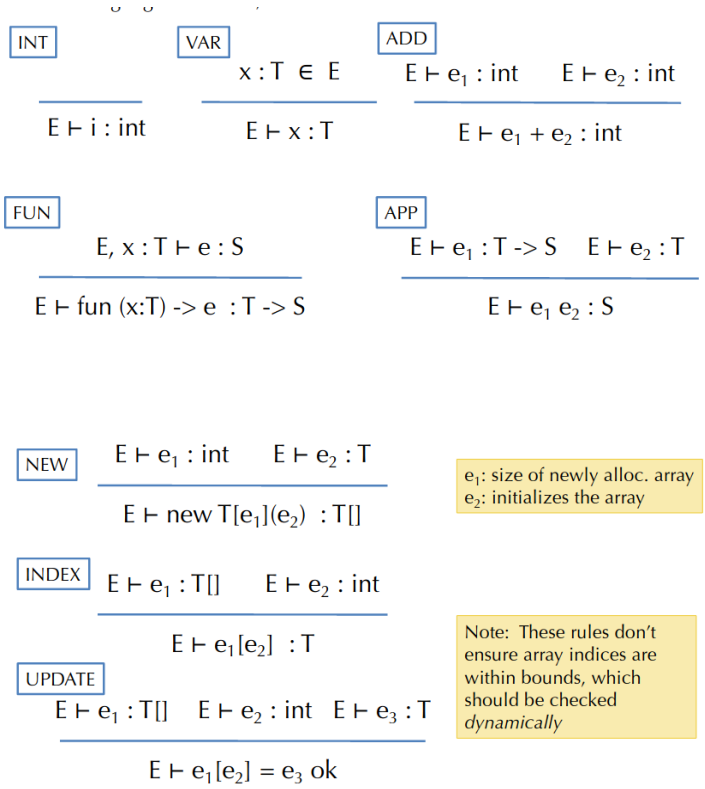
Theorem (in simply typed lambda calculus with integers):
If $\vdash$ `e : t`, then there exists a value `v` such that `e` $\Downarrow$ `v`.

This is a very strong property. Well-typed programs never execute undefined code. Simply-typed lambda calculus terminates (i.e. not Turing complete).

Theorem Type Safety (in general):
If $\vdash$ `P : t` is a well-typed program, then either
a) the program terminates in a well-defined way, or
b) the program continues computing forever.

5.5.2 Type Checking

→ all typing rules in lecture slides (from lec15 onwards)



TUPLE

$$\frac{E \vdash e_1 : T_1 \quad \ldots \quad E \vdash e_n : T_n}{E \vdash (e_1, \ldots, e_n) : T_1 * \ldots * T_n}$$

PROJ

$$\frac{E \vdash e : T_1 * \ldots * T_n \quad 1 \leq i \leq n}{E \vdash \#i\ e\ :\ T_i}$$

For references: add a new type constructor T ref

REF

$$\frac{E \vdash e : T}{E \vdash \text{ref } e : T \text{ ref}}$$

DEREF

$$\frac{E \vdash e : T \text{ ref}}{E \vdash !e\ : T}$$

ASSIGN

$$\frac{E \vdash e_1 : T \text{ ref} \quad E \vdash e_2 : T}{E \vdash e_1 := e_2\ : \text{unit}}$$

typing rules for "if"

$$\text{IF-T}\ \frac{E \vdash e_1 : \text{True} \quad E \vdash e_2 : T}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T} \qquad \text{IF-F}\ \frac{E \vdash e_1 : \text{False} \quad E \vdash e_3 : T}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T}$$

Figure 3: Two cases are easy

But what if we don't know statically which branch will be taken?
⇒ type is the *least upper bound* of the two possible branches/types.

$$\text{IF-BOOL}\ \frac{E \vdash e_1 : \text{bool} \quad E \vdash e_2 : T_1 \qquad E \vdash e_3 : T_2}{E \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : \mathbf{LUB(T_1,T_2)}}$$

Subtyping
Note: the subtyping relation is a partial order: reflexive, transitive, anti symmetric.
A subtyping relation <: is *sound* if it approximates the underlying semantic subset.
Formally: write $[\![T]\!]$ for the subset of (closed) values of type T. If T_1 <: T_2 implies $[\![T_1]\!] \subseteq [\![T_2]\!]$, then <: is sound.

⇒ use this for our typing rules.
Subsumption rule for changing a type to it's "supertype".

Downcasting
add checked downcast

$$\frac{E \vdash e_1 : \mathsf{Int} \qquad E, x : \mathsf{Pos} \vdash e_2 : T_2 \qquad E \vdash e_3 : T_3}{E \vdash \mathsf{ifPos}\ (x = e_1)\ e_2\ \mathsf{else}\ e_3 : T_2 \vee T_3}$$

`T2 V T3` := LUB(T2,T3), "join"-operator

Now: if-rule revisited.
Type of `if (e1) e2 else e3` := `LUB(T1,T2)`

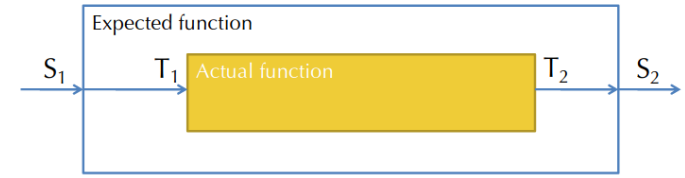
$$\frac{T_1 <: S_1 \qquad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

Figure 4: Subtyping tuples

$$\frac{T_1 <: S_1 \qquad T_2 <: S_2}{(T_1 * T_2) <: (S_1 * S_2)}$$

Figure 5: Subtyping functions

One way to see it



$$\text{RECORD}\quad \frac{E \vdash e_1 : T_1 \quad E \vdash e_2 : T_2 \quad \ldots \quad E \vdash e_n : T_n}{E \vdash \{lab_1 = e_1;\ lab_2 = e_2;\ \ldots\ ;\ lab_n = e_n\} : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}$$

$$\text{PROJECTION}\quad \frac{E \vdash e : \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\}}{E \vdash e.lab_i : T_i}$$

Figure 6: Subtyping immutable records (structs)

- **Depth subtyping**
  - Corresponding fields may be subtypes

$$\text{DEPTH}\quad \frac{T_1 <: U_1 \quad T_2 <: U_2 \quad \ldots \quad T_n <: U_n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}U_1;\ lab_2{:}U_2;\ \ldots\ ;\ lab_n{:}U_n\}}$$

- **Width subtyping**
  - Subtype record may have **more** fields

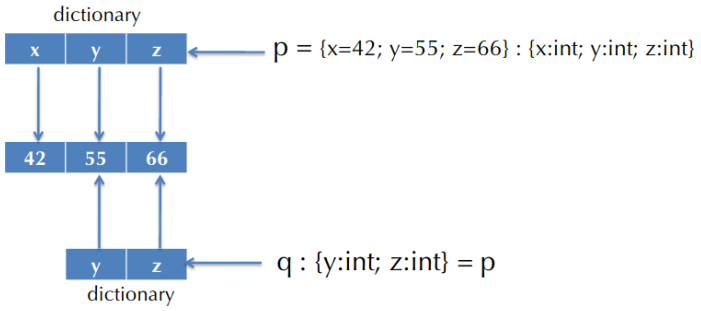
$$\text{WIDTH}\quad \frac{m \leq n}{\{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_n{:}T_n\} <: \{lab_1{:}T_1;\ lab_2{:}T_2;\ \ldots\ ;\ lab_m{:}T_m\}}$$

Figure 7: Depth/Width subtyping for record fields

Width subtyping (without depth) is compatible with "inlined" record representation as with C structs.
Depth subtyping (without width) is similarly compatible, assuming that the space used by A is the same as the space used by B whenever A <: B.
But they don't mix well

Width subtyping assumes implementation where order of fields matters. Alteratively, can allow permutation of records fields. But then the compiler needs to sort the fields before code generation, needs to know all the fields to generate the code. Permutation is not directly compatible with width subtyping.

If we want both permutability and field dropping, we need to either copy (to rearrange the fields) or use a dictionary like the following:



- We can think of a reference cell as an immutable record (object) with two functions and some hidden state:
  T ref ≃ {get: unit -> T; set: T -> unit}
  - get returns the value hidden in the state
  - set updates the value hidden in the state

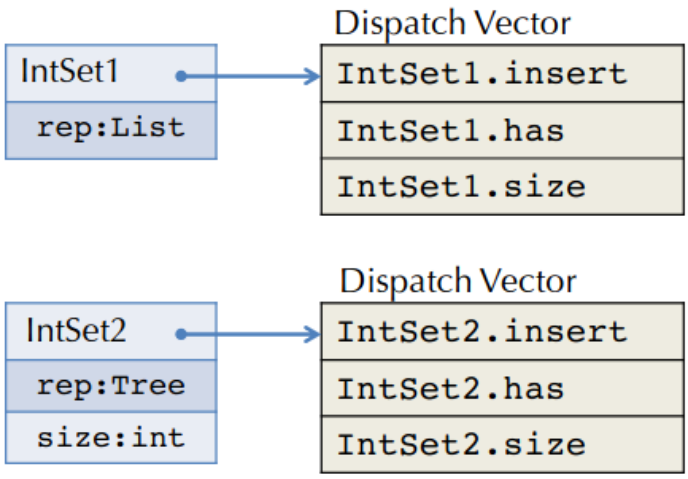
- When is **T ref <: S ref** ?
- Records, like tuples, subtyping extends pointwise over each component
- {get: unit -> T; set: T -> unit} <: {get: unit -> S; set: S -> unit}
  - get components are subtypes: unit -> T <: unit -> S
    set components are subtypes: T -> unit <: S -> unit
- From get, we must have T <: S (covariant return)
- From set, we must have S <: T (contravariant arg.)
- From T <: S and S <: T we conclude T = S

Figure 8: Subtyping reference types

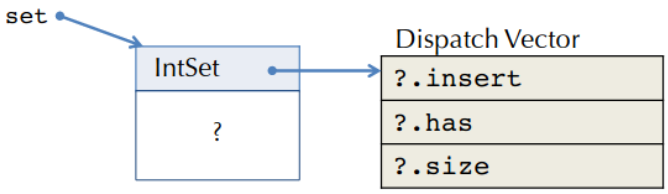
5.6 The dispatch problem/Inheritance

Consider two classes that implement the same interface, i.e. the instances will have the same methods names but different implementations.
Objects must "know" which code to call, which method they implement.

Object contain a pointer to a *dispatch vector* (also called *virtual table/vtable*) with pointers to method code.



Code receiving an object `set:IntSet` only knows that `set` has an initial dispatch vector pointer and the layout of that vector.



Every method has its own small integer index. Index is used to look up the method in the dispatch vector.
Each interface has its own layout, where the inherited methods have the same indices in the subclass (width subtyping).

5.7 Multiple inheritance

Problem: ambiguity. What if a class extends two other classes that both have the same methods? Which to implement?
Also: which methods get which indices in the dispatch vector?
Cannot directly identify methods by positions anymore.

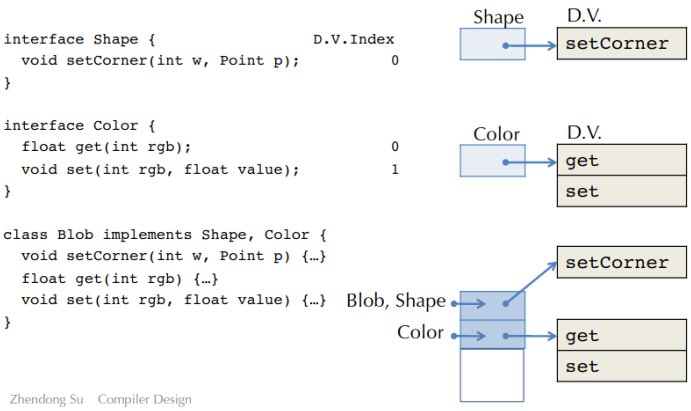
Option 1: Allow multiple D.V. tables (C++)
choose which DV to use based on static type
casting from/to a class may require runtime operations

Option 2: Use a level of indirection
Map method identifiers to code pointers (e.g. indexed by method name)
Use a hash table
May need to search up the class hierarchy

Option 3: Give up separate compilation
Use "sparse" dispatch vectors, or binary decision trees
Must know the entire class hierarchy

5.7.1 Option 1: Multiple D.V.

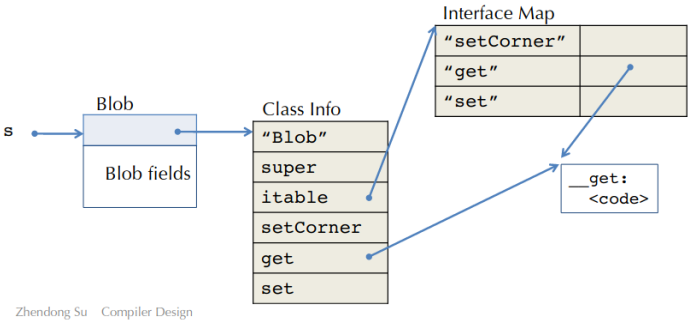
Duplicate D.V. pointers in the object representation. Static type of object determines which D.V. is used.



Pros:
efficient dispatch, similar cost as for a single inheritance
Cons:
Cast has a runtime cost
More complicated programming model, hard to understand/debug

5.7.2 Option 2: Search + Inline Cache

for each class/interface, keep a table: *method names* $\rightarrow$ *method code*
Recursively walk up the hierarchy looking for the method name



Optimization: at call site, store class and code pointer in a cache. On method call, check whether class matches cached value.

Other variant for this option is using a **hash table**. Don't try to give all methods unique indices. Resolve conflicts by checking that the entry is correct at dispatch.
Use hashing to generate indices. Range of hash values should be relatively small. Hash indices can be precomputed, but passed as an extra parameter.

What if there is a conflict?
Entries containing several methods point to code that resolves conflict. e.g. by searching through a table based on class name.

Pros:
Simple, basic code dispatch is almost identical. Reasonably efficient
Cons:
Wasted space in DV. Extra argument needed for resolution. Slower dispatch if conflict

5.7.3 Option 3: Sparse DV tables

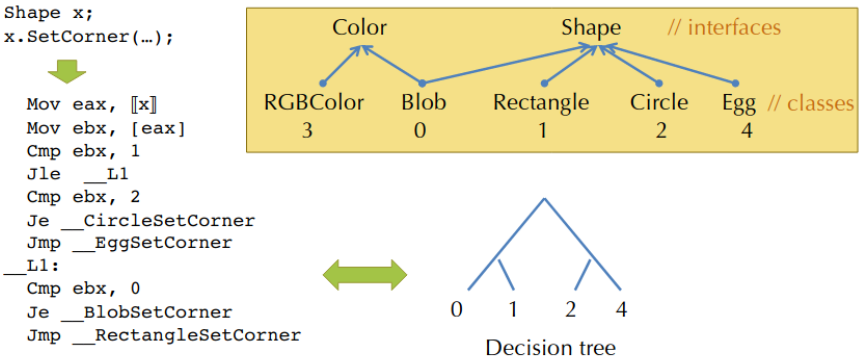
Variant 1:
Give up on separate compilation. Now we have access to the whole class hierarchy.
Ensure no 2 methods in same class are allocated the same D.V. offset. Allow holes in the D.V. Unlike hash table, there will never be a conflict.

Compiler needs to construct the method indices. Graph coloring can be used to construct D.V layouts reasonably efficient. finding optimal is NP complete.

Pros:
identical dispatch and performance to single inheritance case
Cons:
Must know entire class hierarchy.

Variant 2: Binary Search Trees
Each object has a class index (unique per class) as first word, instead of a D.V. pointer. Method invocation uses range tests to select among n possible classes in $\log n$ time. Direct branches to code at the leaves.

Pros:
Work well if the distribution of classes that may appear at a call site is skewed. Profiling helps find the common paths for each call site individually (put the common case at the top of the decision tree.
Cons:
need the whole class hierarchy to know how many leaves are needed in search tree. indirect jumps can have better performance if there are >2 classes (at most one mispredict)



5.8 Representing classes in LLVM

- LLVM IR struct type for each object instance
- LLVM IR struct type for each vtable (aka class table)
- Global definitions that implement the class table

Method bodies are compiled just like top-level procedures, except that they have an implicit extra argument: `this` or `self`. The type of `this` is the class containing the method. References to fields inside the body are compiled like `this.field`.

5.8.1 LLVM method invocation compilation

Consider method invocation ⟦`H;G;L ⊢ e.m(e1, ..., en):t`⟧
In general, function calls may require a bitcast to account for subtyping.

1. Compile ⟦ `H;G;L ⊢ e : C` ⟧ to get a pointer to an object value of class type `C`. Call this value `obj_ptr`.
2. Use `getelementptr` to extract the vtable pointer from obj_ptr
3. Load the vtable pointer
4. Use `getelementptr` to extract the function pointer's address from vtable. Use information about `C` in `H`
5. Load the function pointer
6. Call through the function pointer, passing `obj_ptr` for this:
   `call (cmp_typ t) m(obj_ptr, ⟦e1⟧, ..., ⟦en⟧`

In general, function calls may require a bitcast to account for subtyping.

All instances of a class may share the same D.V.

5.8.2 Compiling static methods

Just like normal functions; they are not really methods

5.8.3 Compiling constructors

Compiled like static methods, except the "this" variable is initialized to a newly allocated block of memory big enough to hold D.V. pointer + fields according to object layout.
Constructor code initializes all the fields, D.V. pointer is initialized.

6 Basic optimizations

Safe vs. unsafe optimizations: whether they change the meaning/behavior of the program

Overview:

- Inlining
- Function specialization
- Constant folding
- Constant propagation
- Value numbering
- Dead code elimination
- Loop-invariant code motion
- Common sub-expression elimination
- Strength reduction
- Constant folding & propagation
- Branch prediction / optimization
- Register allocation
- Loop unrolling
- Cache optimization

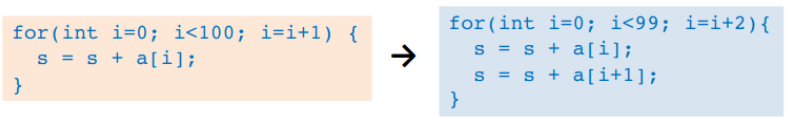


Figure 9: Example: Loop unrolling

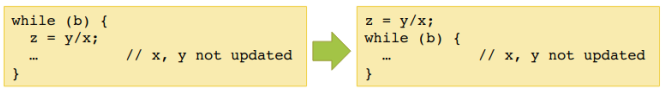


Figure 10: Example: Loop invariant code motion

Constant folding
Idea: if operands are statically known, compute value at compile-time. Can be performed at every stage of optimization.
Can be done with arithmetic operations, conditionals, ...

→ all forms of algebraic simplification.
Also includes e.g. strength reduction, replacing expensive ops with cheaper ones (`a * 4` → `a « 2`).

Constant propagation
Replace occurrences of a variable with constant value with that value. Interleave const. prop and folding.
In addition: *Copy propagation*: if variable y is assigned to x, replace x's uses with y.
Can make the first assignment of x *dead code*, thus eliminated.

Dead code elimination
If side-effect free code can never be observed, safe to eliminate it.
e.g.

```
x = y * y
...  // x is never used
x = z * z
```

A variable is dead if it is never used after it's defined. Dead variables can be created by other optimizations!

Can also check for dead blocks, blocks unreachable from entry block

Inlining
Replace a function call with the body of the function (with arguments rewritten to be local variables).
May need to rewrite variable names to avoid name capture. Best done at the AST or relatively high level IR.
Enables further optimization, eliminates stack manipulation and jumps, but increases code size.

Code specialization
Create a specialized version of a function that is called from different places with different arguments.

Common subexpresion elimination
In some sense the opposite of inlining: fold redundant computations together.
e.g.
`a[i] = a[i] + 1` compiles to

`[a + i*4] = [a + i*4] + 1`
optimize → `t = a + i*4; [t] = [t] + 1`

When is it safe? The shared expression must always have the same value in both places!

Loop invariant code motion
If the result of a statement or expression does not change during the loop and it has no external effects, it can be hoisted outside the loop body.

Strength reduction (for loops)
Create a dependent induction variable:
`for (int i = 0; i<n; i++) { a[i*3] = 1; } // stride 3`
⇒ `int j = 0;`
`for (int i = 0; i<n; i++) { a[j] = 1; j = j + 3; }`

Loop unrolling
Since branches can be expensive, unroll loops to avoid them.
With k unrollings, eliminates (k-1)/k conditional branches.

7 Dataflow Analysis

• **Instructions n**	**def[n]**	**use[n]**	**description**
a = op b c	{a}	{b,c}	arithmetic
a = load b	{a}	{b}	load
store c, b	Ø	{b}	store
a = alloca t	{a}	Ø	alloca
a = bitcast b to u	{a}	{b}	bitcast
a = gep b [c,d, …]	{a}	{b,c,d,…}	getelementptr
a = f(b_1,…,b_n)	{a}	{b_1,…,b_n}	call w/return
f(b_1,…,b_n)	Ø	{b_1,…,b_n}	void call (no return)
• **Terminators**			
br L	Ø	Ø	jump
br a L1 L2	Ø	{a}	conditional branch
return a	Ø	{a}	return

Figure 11: def/use for SSA

7.1 Forward vs. Backward, May vs. Must

Forward: `in[n]` computed based on `out[n]`, Backward: `out[n]` computed based on `in[n]`

May: take union ∪ → a property that holds for *some* path, Must: take intersection ∩ → a property that holds for *all* paths.

- Liveness: **(backward, may)**
  - Let gen[n] = use[n] and kill[n] = def[n]
  - out[n] := $\bigcup_{n' \in succ[n]}$ in[n']
  - in[n] := gen[n] ∪ (out[n] \ kill[n])
- Reaching Definitions: **(forward, may)**
  - in[n] := $\bigcup_{n' \in pred[n]}$ out[n']
  - out[n] := gen[n] ∪ (in[n] \ kill[n])
- Available Expressions: **(forward, must)**
  - in[n] := $\bigcap_{n' \in pred[n]}$ out[n']
  - out[n] := gen[n] ∪ (in[n] \ kill[n])

Figure 12: (backward, must) would be Very Busy Expressions

	May	**Must**
Forward	Reaching definitions	Available expressions
Backward	Live variables	Very busy expressions

7.1.1 General framework

A forward dataflow analysis can be characterized by

1. A domain of dataflow values $\mathcal{L}$
2. For each node n, a flow function $F_n : \mathcal{L} \to \mathcal{L}$
   e.g. $F_n(l) =$ `gen[n]` $\cup$ `(` $l \setminus$ `kill[n]`
   `out[n] =` F_n`(in[n])`
3. A combining operator $\sqcap$
   `in[n] :=` $\bigsqcap_{n' \in pred[n]}$ `out[n']`

Generic iterative forward analysis:

```
for all n, in[n] := ⊤, out[n] := ⊤
repeat until no change:
for all n:
in[n] := ⊓_{n'∈pred[n]} out[n']
out[n] := F_n(in[n])
```

where $\top$ = "top", the "maximum" amount of information

The meet is the greatest lower bound:

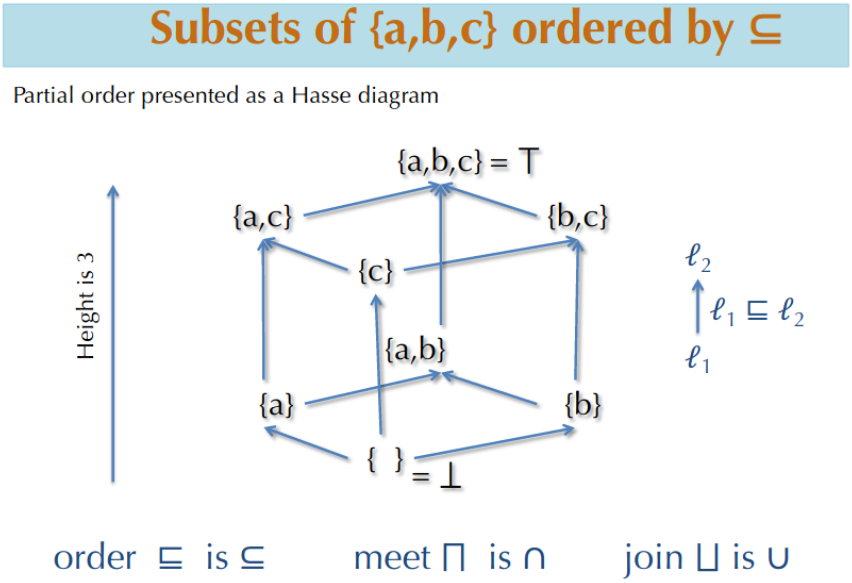


Figure 13: Partially ordered subsets of {a,b,c}

$l_1 \sqcap l_2 \subseteq l_1$ and $l_1 \sqcap l_2 \subseteq l_2$

The join is the least upper bound:
$l_1 \subseteq l_1 \sqcup l_2$ and $l_2 \subseteq l_1 \sqcup l_2$

A partial order that has all meets and joins is called a lattice.

7.2 Liveness

(backward, may)
Domain: set of variables
Liveness: a variable is called live if its contents will be used as a source operand in a later instruction.
Two variables can share a register if they are not live at the same time.

Variable scope as an indicator for liveness is too coarse.

define `use[s]`: set of variables used by s
and `def[s]`: set of variables defined by s

A variable is *live* on edge e if
- there is a node n in the CFG such that `use[n]` contains v,
- and there is a directed path from e to n such that for every statement s' on the path, def[s'] does not contain v

define `in[n]`, `out[n]`: sets of variables that are live on entry/exit to/from n

Constraints:
`use[n]` $\subseteq$ `in[n]`
`(out[n]` $\setminus$ `def[n])` $\subseteq$ `in[n]`
`in[n']` $\subseteq$ `out[n]` if `n'` $\in$ `succ[n]`

$\Rightarrow$ iterative dataflow analysis
`for all n, in[n] :=` $\varnothing$`, out[n] :=` $\varnothing$
`repeat until no change in 'in' and 'out':`
`for all n:`
`out[n] :=` $\cup_{n' \in succ[n]}$ `in[n']`

Algorithm is guaranteed to terminate.

Improved algorithm: if a node n's successors haven't changed, n won't change.
$\Rightarrow$ use a FIFO queue of nodes that might need to be updated.

```
for all n, in[n] := ∅, out[n] := ∅
w = new queue with all nodes
repeat until w is empty:
let n = w.pop()
old_in = in[n]
out[n] := ∪_{n'∈succ[n]} in[n']
in[n] := use[n] ∪ (out[n] - def[n])
if (old_in != in[n]) { for all m in pred[n]:  w.push(m) }
```

7.3 Reaching Definitions

(forward, may)
Domain: set of nodes
What variable definitions reach a particular use of the variable? used for constant propagation, copy propagation.
Define sets of interest for the analysis: `gen[n]` and `kill[n]`
Constraints:

Quadruple forms n	gen[n]	kill[n]
a = b op c	{n}	defs[a] \ {n}
a = load b	{n}	defs[a] \ {n}
store b, a	Ø	Ø
a = f(b_1,…,b_n)	{n}	defs[a] \ {n}
f(b_1,…,b_n)	Ø	Ø
br L	Ø	Ø
br a L1 L2	Ø	Ø
return a	Ø	Ø

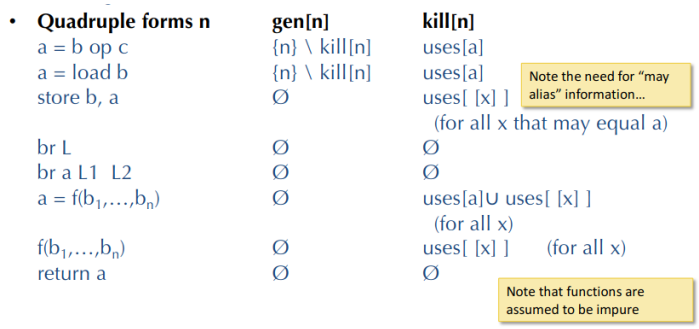
`gen[n]` $\subseteq$ `out[n]`
`out[n']` $\subseteq$ `in[n]` if `n' is in pred[n]`
`in[n]` $\subseteq$ `out[n]` $\cup$ `kill[n]`

Convert constraints to iterated update equations:
`in[n] := ` $\cup_{n' \in pred[n]}$ `out[n']`
`out[n] := gen[n] ∪ (in[n] \ kill[n])`

Initialize `in[n], out[n]` to $\varnothing$. Iterate the update equations until a fixed point is reached.
Algorithm terminates and is precise (finds smallest set satisfying the constraints)

7.4 Available Expressions

(forward, must)
Domain: set of nodes
Want to perform common subexpression elimination → identify parts of the code that compute the same value.
Constraints:



`gen[n]` $\subseteq$ `out[n]`
`in[n]` $\subseteq$ `out[n']` if `n' is in pred[n]`
`in[n]` $\subseteq$ `kill[n]` $\cup$ `out[n]`

Convert to iterated update equations:
`in[n] := ` $\cap_{n' \in pred[n]}$ `out[n']`
`out[n] := gen[n] ∪ (in[n] \ kill[n])`

Initialize `in[n], out[n]` to the set of all nodes, iterate until fixed point is reached.
Terminates and is precise.

7.5 Very Busy Expressions

(backward, must)
An expression e is very busy at location p if every path from p must evaluate e before any variable in e is redefined

7.6 Constant Propagation

For multiple variables, take the product lattice, with one element per variable: variables x, y, z ⇒ elements of the form (l_x, l_y, l_z).

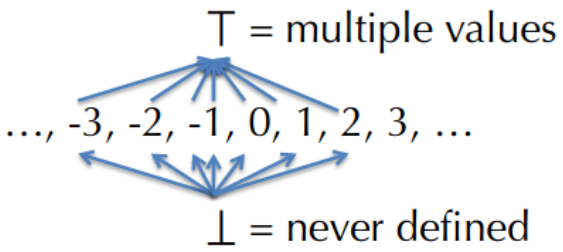


Figure 14: lattice for one variable

What is the flow function? Consider the node `x = y op z`
$F(l_x, \top, l_z) = (\top, \top, l_z)$: "if either input might have multiple values, the result of the operation might too."
$F(l_x, \bot, l_z) = (\bot, \bot, l_z)$: "if either input is undefined, the result of the operation is undefined too."
$F(l_x, i, j) = (i \texttt{ op } j, i, j)$: "if the inputs are known constants, calculate the output statically."

7.7 MOP solutions

What is the best possible solution for the information at a node n? It is the Meet-over-paths: $\bigsqcap_{p \in paths\_to[n]} l_p$, where $l_p = F_{n_k}(...(F_{n_2}(F_{n_1}(\top))))$

The iterative solutions compute the MOP solution, if the flow function distributes over $\sqcap$.

- Liveness Analysis is MOP
- Available Expressions is MOP
- Reaching Definitions is MOP
- Very Busy Expressions is MOP
  → these are all analyses of *how* the program computes
- Constant Propagation is **not** MOP
  → this is an analysis of *what* the program computes. MOP solution is exponential in n.

SSA = Single Static Assignment → variables cannot change value

8 Register Allocation

8.1 Linear-Scan

1. Compute liveness information `live(x)`
   set of uids that are live on entry to x's definition
2. Let `pal` be the set of usable registers
   usually reserve a couple for spill code (rax, rcx)
3. Maintain "layout" `uid_loc` that maps uids to locations
   locations include registers and stack slots n, starting at n=0
4. Scan through the program, for each instruction that defines a uid x:
   - `used = { r | reg r = uid_loc(y) s.t.  y ∈ live(x) }`
   - `available = pal - used`
   - `if available is empty:  // no registers available, spill`
   `uid_loc(x) := slot n; n=n+1`
   - `Otherwise, pick r in available:  // choose an available register`
   `uid_loc(x) := reg r`

8.2 Graph-Coloring

Smarter approach:

1. Compute liveness information for each temp
2. Create an interference graph
   - nodes are temps
   - there is an edge between nodes n and m if they are live at the same time
3. Try to color the graph
   Each color corresponds to a register
4. if step 3 fails, "spill" a register to the stack and repeat form step 1
5. rewrite the program to use registers

Kempe's Algorithm
Recursive algorithm for K-coloring a graph

1. Find a node with degree $< K$ and cut it out of the graph
   - remove the node and edges (called *simplifying* the graph)
2. Recursively K-color the remaining subgraph
3. When remaining graph is colored, there must be at least one free color available for the deleted node (since its degree was $< K$). Pick such a color.

The algorithm can fail even if the graph is K-colorable!

8.2.1 Spilling

If we can't K-color the graph, need to store 1 temp on the stack.

Which variable to spill?
- Pick one that isn't very frequently used
- pick one that isn't used in a (deeply nested) loop
- pick one that has high interference (will make coloring the rest of the graph easier)

⇒ mark the node as spilled, remove it from the graph, keep recursively coloring.

8.2.2 Optimistic coloring

Don't spill nodes directly, only mark them for spilling. When coming out of the recursion, we might get lucky and still be able to assign it a color.

8.2.3 Generating code for spilling

Option 1:
Reserve registers specifically for moving to/from memory.

Option 2:
Rewrite the program to use a new temp with explicit moves to/from memory
→ must recompute liveness & recolor graph.

8.2.4 Precolored nodes

Some variables must be pre-assigned to registers, e.g. multiplication instruction on X86 must define %rax; call instruction should kill caller-save registers %rax, %rcx, %rdx; any temp live across a call interferes with the caller-save registers.

→ pre-assign colors to these nodes in the graph. They can't be removed during simplification (trick: treat them as having infinite degree in the interference graph). When the graph is empty except the pre-colored nodes, we have reached the point where we start coloring the rest of the nodes.

8.2.5 Picking good colors

any choice is semantically correct, but can be optimized. E.g.
Choose colors to minimize move operations: `movq t1, t2`: if t1 and t2 can be assigned the same color (register), this move is redundant and can be eliminated.
→ Add a new kind of move-related edge between nodes t1 and t2 in the interference graph. When choosing a color for t1 or t2, if possible pick a color of an already colored node reachable by a move-related edge.

8.2.6 Coalescing interference graphs

More aggressive strategy: coalesce nodes of the interference graph if they are connected by a move-related edge (combine the nodes into one). Interleave simplification and coalescing to maximize the number of moves that can be

eliminated. Problem: coalescing may increase the degree of the node.

Two strategies are guaranteed to preserve the k-colorability of the graph:
Brigg's strategy
it's safe to coalesce x & y if the resulting node will have fewer than k neighbors that have degree >= k
- The merged node (x,y) can still be removed.
George's strategy
We can safely coalesce x & y if for every neighbor t of x, either t already interferes with y or t has degree < k.
- Let S be the set of neighbors of x with degree < k
- If no coalescing, simplify and remove all nodes in S ⇒ G1
- If we coalesce, can still remove all ndoes in S ⇒ G2
- G2 is a subgraph of G1

For Briggs, ned to look at all neighbors of x and y. For George, we need to look at only the neighbors of x. Precolored nodes have infinite degree.
Thus, we
- use George's strategy if one of x and y is precolored
- use Briggs' strategy if both are temps

8.2.7 Complete register allocation algorithm

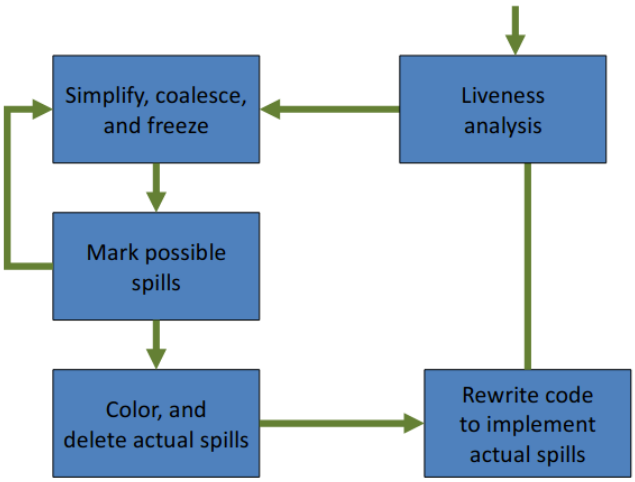
1. Build interference graph (precolor nodes as necessary)
   - Add move-related edges

2. Reduce the graph (building a stack of nodes to color)
   - Simplify the graph as much as possible without removing nodes that are move-related. Remaining nodes are high degree or move-related
   - Coalesce move-related nodes using Briggs' or George's strategy
   - Coalescing can reveal more nodes that can be simplified, so repeat 2.1 and 2.2 until no node can be simplified or coalesced
   - If no nodes can be coalesced *freeze* (remove) a move-related edge and keep trying to simplify/coalesce

3. If there are non-precolored nodes left, mark one for spilling, remove it from the graph and continue doing step 2

4. When only precolored nodes remain, start coloring (popping simplified nodes off the top of the stack)
   - If a node must be spilled, insert spill code as shown earlier and rerun the whole register allocation algorithm starting at step 1

After register allocation, the compiler should do a peephole optimization pass to remove redundant moves



9 Control-Flow Analysis

9.1 Dominator Trees

A dominates B: if the only way to reach B from the start node is via A.
An edge in the CFG is a *back edge*, if its target dominates the source.

Domination is transitive and anti-symmetric (A dom B & B dom A $\Rightarrow$ A = B).

We can define `Dom[n]` as a forward dataflow analysis: the set of all nodes that dominate n.
`Dom[n] = out[n]` where B is dominated by A if A dominates all of B's predecessors
`in[n] :=` $\bigcap_{n' \in pred[n]}$ `out[n']`

Every node dominates itself: `out[n] := in[n]` $\cup$ `{n}`

Formally:
$\top$ = { all nodes }
F_n`(x)` = `x` $\cup$ `{n}`
$\sqcap = \cap$

$\Rightarrow$ terminates, distributes, computes MOP

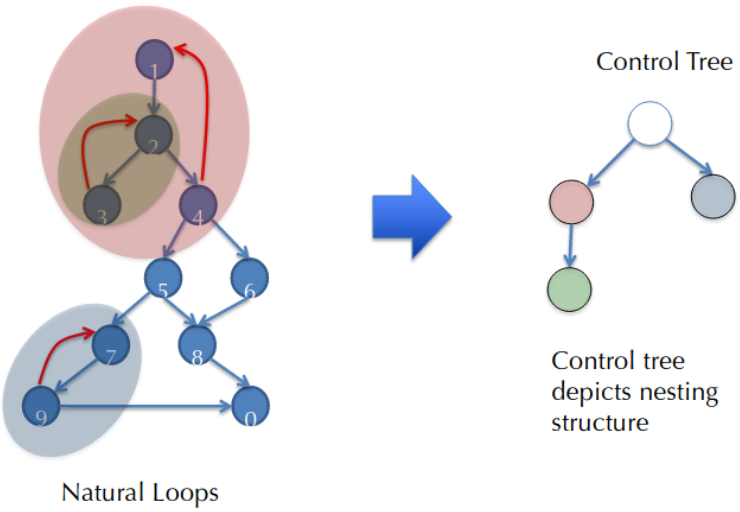


Figure 15: Natural loops: created through back edges, can be identified based on dominance and used to turn into control tree.

9.1.1 Improving the algorithm

Since there is a lot of sharing between the Dom sets (because of transitivity), it would be more efficient to represent Dom sets by storing the dominator tree: `doms[b]` = immediate dominator of b. Then `Dom[n]` is a walk through `doms[b]`.

9.2 Dominance Frontiers

Node A **strictly dominates** B if: A dom B & A $\neq$ B

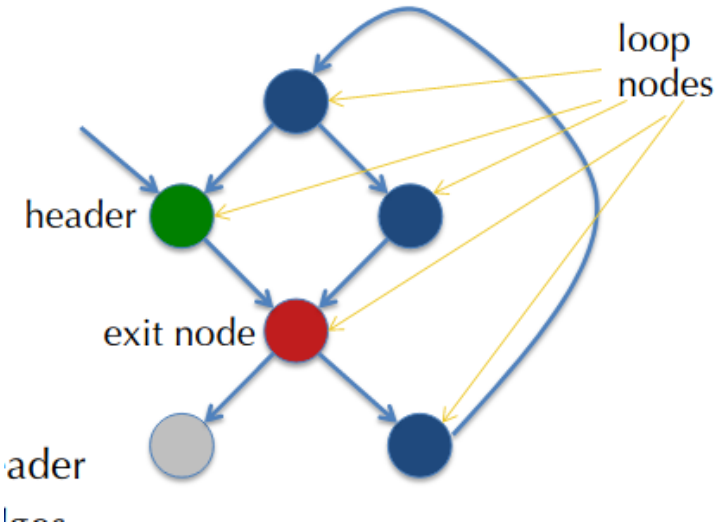
The **dominance frontier** of a node B is the set of all nodes y such that B dominates a predecessor of y, but does not strictly dominate y.
Intuitively: There is a path from B to y, but there is also another route to y that does not go through B.

9.2.1 Algorithm for computing DF[n]

Adds each B to the DF sets to which it belongs:

for all nodes B:
.. if #(pred[B]) $\geq$ 2:
.... for each p $\in$ pred[B]:
...... runner := p
...... while (runner $\neq$ doms[B]):
.... DF[runner] := DF[runner] $\cup$ {B}
.... runner := doms[runner]

9.3 Loops



Definition: A set of nodes in the CFG with one distinguished entry node, the **header**. Each node is reachable from the header, and the header is reachable from each node. No edges enter a loop except to header. There are also **exit nodes**, nodes with outgoing edges. Also, a loop contains at least one back edge. A loop is a strongly connected component.

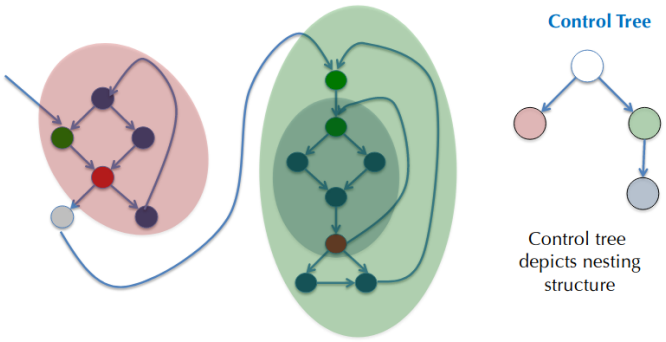


Figure 16: Loops can also be nested, i.e. a loop may contain other loops.

10 SSA & Phi nodes

to be done
Lec21

11 Garbage Collection

If we want type safety, we typically must use automatic memory management (GC).

Basic assumption: A program can only use objects that it can find. Memory for the rest can be freed.
An object x is **reachable** iff - a register contains a pointer to x, or
- another reachable object y contains a pointer to x

⇒ start from registers, follow all the pointers.

11.1 Mark & Sweep

Every object has an extra bit: mark bit, which is initially 0.

Mark phase
Follow all the pointers, and set the mark bit of all found objects to 1.

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

Figure 17: the mark phase

Sweep phase
Scan the heap for objects with mark bit 0. These objects are garbage. Add them to the *free list*. Reset mark bit of all objects to 0.

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

Figure 18: the sweep phase

While conceptually simple, there are some tricky details.
Mark phase is invoked when we are out of space, yet it needs space to construct the todo list. Size of todo list is unbounded, so cannot reserve space a priori.

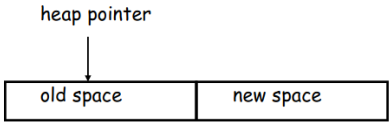
Use **pointer reversal**: when a pointer is followed, reverse it to its parent.
Similarly, the free list is stored in the free objects themselves.

Pros:
Objects are not moved during GC

Cons:
Mark and sweep can fragment the memory

11.2 Stop & Copy

Memory is organized into two areas: old space, used for allocation, and new space, used as a reserve for GC.

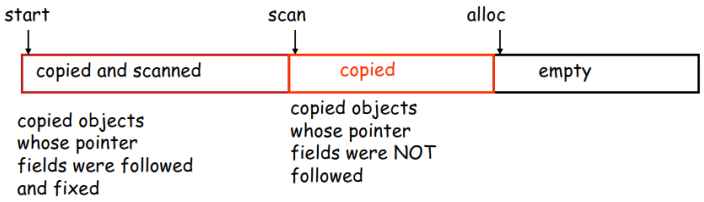


The heap pointer points to the next free word in old space.

When old space is full, copy all reachable objects from old space to new space.
Garbage is left behind → after copy phase, new space uses less space than old space before GC.
Now the roles of old and new space are reversed, and the program resumes.

As we copy an object, we have to fix *all* pointers to it!
- Store a forwarding pointer to the new copy in the place of the old copy
- Any object reached later with a forwarding pointer was already copied

To implement the traversal w/o using extra space, partition the new space into three contiguous regions:



After copying an object, follow the pointers it contains, copy those objects and fix the pointers of the object pointing to it.

When scan pointer reaches alloc pointer, we are done!

```
while scan <> alloc do
    let O be the object at scan pointer
    for each pointer p contained in O do
        find O' that p points to
        if O' is without a forwarding pointer
            copy O' to new space (update alloc pointer)
            set 1st word of old O' to point to the new copy
            change p to point to the new copy of O'
        else
            set p in O equal to the forwarding pointer
        fi
    end for
    increment scan pointer to the next object
od
```

Figure 19: the stop and copy algorithm

Pros:
- believed to be the fastest GC technique
- allocation is very cheap
- collection is relatively cheap

Cons:
Need to copy everyting

11.3 Conservative GC

In general, it's impossible to know what is a pointer and what not (identifying the contents of memory).
But it's ok to be conservative: if a memory looks like a pointer, it is considered a pointer:
- it must be aligned
- it must point to a valid address in the data segment.

All such pointers are followed ⇒ we overestimate reachable objects

11.4 Reference Counting

Rather than waiting for memory to run out, try to collect an object when there are no more pointers to it.
Store in each object the number of pointers to that object ⇒ the reference count

`new` returns an object with a reference count (rc) of 1
For every assignment `x := y`, we must change:

```
rc(y) ← rc(y) + 1
rc(x) ← rc(x) - 1
if(rc(x) == 0) then mark x as free
x := y
```

Pros:
- easy to implement
- collects garbage incrementally without large pauses in execution

Cons:
- Manipulating reference counts at each assignment is very slow
- need to handle circular reference structures (slide says "cannot collect circular structures")