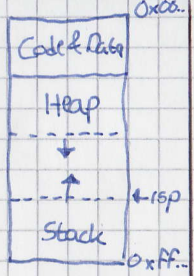
# Compiler Design

**X86-Lite**: Rsp = Top of stack

Heap: Stores dynam. alloc. Objects

Stack: Stores local vars & return addr.

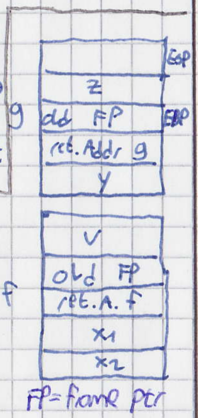



**Calling Convention** SystemV AMD64 ABI:

- Callee save: rbp, rbx, r12-r15
- Params 1..6: rdi, rsi, rdx, rcx, r8, r9
  - 7+: on stack, right-to-left

**Function Call Frames:**

$f(x_1, x_2)$ with local variable $v$ calls $g(y)$ with local variable $z$




**Basic Block**: Seg. of instr. that execute together, starting at first instr. & end at last instr.

## Compiler Structure

Src Code → Lexical Analysis $\xrightarrow{\text{Token Strm}}$ Parsing $\xrightarrow{\text{AST}}$ Intermed. Code Generation $\xrightarrow{\text{Intermed. Code}}$ Code Generation → Assembly Code

## Lexing

„Character Stream" → Tokens

**Token** is a datatype that represents „chunks" of text, e.g. Identifiers, Keywords, Integers etc.

**Regex** R:

| | |
|---|---|
| ε → Empty str | $R^*$ → zero or more |
| 'a' → Char. "a" | $R^+$ → one or more |
| $R_1 \| R_2$ → either of | R? → zero or one |
| $R_1 R$ → $R_1$ followed by $R_2$ | ['a'-'z'] → Any in Range |
| | [^'0'-'9'] → Any except. in Range |

## Lexer Generator

- Reads list of Regex $R_1, \dots, R_n$, one per Token
- Each token has an „Action" $A_i$ (piece of code to run when $R_i$ is matched)

**DFA for Lexing**: Regex can be represented by a DFA (Graph or Transition Table Represent.)

**NFA for Lexing**: Has states & transitions like DFA, can also have trans. lbl ε which does not consume input. Two arrows leaving the same state may have same lbl (nondet.)

## Lexer Behaviour

1) Take each Regex $R_i$ with Action $A_i$
2) Compute NFA formed by $(R_1 | \dots | R_n)$
3) Compute DFA for this NFA
4) Compute minimal equivalent DFA
5) Produce transition table
6) Implement longest match

**Parsing**: Transforms a stream of tokens into an abstract syntax tree (AST)

- Strategy: Parse token stream to traverse „concrete" syntax; during traversal build a tree representing the „abstract" syntax

## Context-free grammars (CFG)

- Terminals (e.g. Lexical token)
- Set of Nonterminals
- Designated Nonterminal called start symbol
- Set of productions LHS → RHS

**Parse-Tree** is a tree representation of the derivation. Leaves are terminals (in-order traversal yields the input sequence.

The internal nodes are the nonterminals

**Leftmost derivation**: Find the leftmost nontermin. & apply a production to it

**Rightmost derivation**: Find the rightmost nonterminal & apply a production to it

## LL(1) Grammar

→ Top-down in tree

- Left-to-right scanning
- Leftmost derivation
- 1 Lookahead symbol

**Remove Left-Recursion**

Rewrite $S \to S\alpha_1 | \dots | S\alpha_n | \beta_1 | \dots | \beta_m$ as $S \to \beta_1 S' | \dots | \beta_m S'$ & $S' \to \alpha_1 S' | \dots | \alpha_n S'$

**First-Set**

First(X) for a grammar symbol X is the set of terminals that begin the strings derivable from X

(Also need to recursively derive non-terminals if they are first in a production.)

**Follow-Set**

$\text{Follow}(A) = \{t \in T \mid S \to \dots \to \beta A t \gamma\}$

↳ Contains all terminals that follow on A.

1) Follow(S) += {\$} (S = Start, \$ = EOL)
2) $A \to \alpha B \beta \Rightarrow \text{Follow}(B) \mathrel{+}= \text{First}(\beta) \setminus \{\varepsilon\}$
3) $A \to \alpha B$ or $A \to \alpha B \beta$ with $\beta \xrightarrow{*} \varepsilon$: Follow(B) += Follow(A)

## LL(1) Parser Table

- Create First & Follow set for all productions. Then:
  for every production $A \to \alpha$:
  1) For every $a$ in First($\alpha$): $\alpha$ in $T[A, a]$
  2) If $\varepsilon$ in First($\alpha$):
     for every terminal $b$ in Follow(A):
     $\alpha$ in $T[A, b]$

## LL(1) Criteria

A grammar is LL(1) iff for 2 different productions $A \to \alpha$ and $A \to \beta$:

1) First($\alpha$) $\cap$ First($\beta$) $= \emptyset$
2) If $\varepsilon \in$ First($\alpha$): First($\beta$) $\cap$ Follow(A) $= \emptyset$
   If $\varepsilon \in$ First($\beta$): First($\alpha$) $\cap$ Follow(A) $= \emptyset$

## Shift / Reduce Parsers

- Parser State:
  - Stack of terminals & non-terminals
  - Unconsumed input is a string of terminals
  - Current derivation step is stack + input

→ Parsing = Shift & Reduce operations

- Shift = Move lookahead token to the stack
- Reduce = Replace symbols $\gamma$ at top of stack with nonterminal $X$, such that $X \to \gamma$ is a production (pop $\gamma$, push $X$)

**LR(0)** Has a state, which is a set of items, keeping track of possible upcoming reductions

→ LR(0) item is a production with seperator „•" somewhere in the RHS. → Intuition:

- Stuff before „•" is on stack (poss. $\gamma$ to be reduced)
- Stuff after „•" is what might be seen next

**LR(0) DFA** Start state: New prod. $S' \to S\$$ with item $S' \to \bullet S\$$.

- Closure of state: Add items for all productions whose LHS nonterminal occurs in an item of the state just after the • (e.g. $S' \to \bullet S\$$: Add all with $S \to \dots$)
- Transitions: Outgoing edges = Terminals & nonterminals that appear after • in src state
- Target state includes all items that have edge symbol after • in src → advance • to simulate shifting on stack
- Reduce state: • at end of rule → reduce when reached

## LR(1) Parsing

state = set of LR(1) items

→ LR(1) item = LR(0) item + set of lookahead symbols

When new item $C \to \bullet\gamma$ is added b/c $A \to \beta \bullet C\delta, L$ (closure), we need to compute new look-ahead set $M$:

1) $M$ includes First($\delta$)
2) If $\delta$ can derive $\varepsilon$: $M$ += $L$

## First class Functions & Lambda calculus

- Write fun $x \to x$ as $\lambda x. x$
- $e\{v/x\}$: Subst. all free $x$ in $e$ by $v$
- Free variable: Defined in an outer scope
- Bound variable: Defined in function

**Alpha Equivalence:** 2 terms that only differ by consistent renaming of bound vars. are called alpha equivalent.

## Inference Rules

- $G; L \vdash e : t$ = „e is well typed and has type t"
- Context: Has bindings like $x$: int, $y$: bool, ...
- exp ⇓ val = „exp evaluates to val"

**Subtyping** $t_1 <: t_2$ = „$t_1$ is subtype of $t_2$"

$$\frac{\vdash t_3 <: t_1 \qquad \vdash t_2 <: t_4}{\vdash t_1 \to t_2 <: t_3 \to t_4} \text{ fun}$$

**Depth Subtyping:** Corresponding fields may be subtypes (number of fields needs to be equal)

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

## Multiple Inheritance

- Dispatch vector: table in class with ptrs to methods

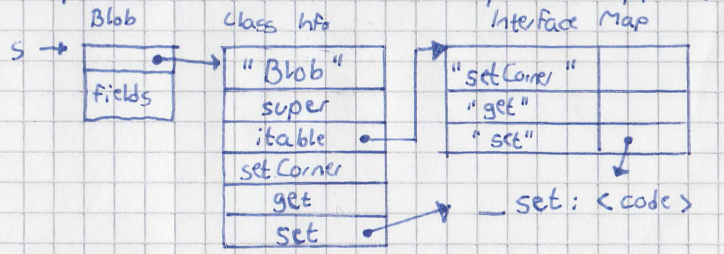
Option 1: **Multiple Dispatch Vectors**

- Choose DV based on static type
- Casting from/to class may require run-time operations

Option 2: **Use a level of indirection**

- Map method identifiers to code pointers
- Use a hash table
- Map need to search up the class hierarchy

Example: Blob s = new Blob() (Blob implements Shape, Color)




Option 2.1: **Use a Hashtable**: Fill Dispatch vector sparsingly, using a hash function. Don't use an interface map.

Option 3: Give up separate compilation
- Get „single class performance"
- Use „sparse" DV or binary decision trees
- Must know the entire class hierarchy

## Register Allocation

Accessing Spilled Registers:
- Option 1: Reserve registers specifically for moving from/to memory
- Option 2: Rewrite the program to use a new temporary variable, with explicit moves from/to mem.

**Kempe's Algorithm**: k Color this Graph
1) Find a node with degree < K and cut it out of the graph → 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. Pick such a color

**Coalescing**: „Merge" nodes of the interference graph if they are connected by move-related edges

**Briggs' Strategy**: It's safe to coalesce x & y if the resulting node will have fewer than k neighbors that have degree >= k

**Georges' 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

**Reaching Definition Analysis**: What variable definitions reach a particular use of the var?

Constraints:
- $out[n] \supseteq gen[n]$
- $in[n] \supseteq out[n']$ if $n'$ in $pred[n]$
- $out[n] \cup kill[n] \supseteq in[n]$

**Available Definitions** $out[n] \supseteq gen[n]$

$in[n] \subseteq out[n']$ if $n'$ in $pred[n]$

$out[n] \cup kill[n] \supseteq in[n]$

## Dataflow Analyses

**Liveness** (backward, may) *

$out[n] := \bigcup_{n' \in succ[n]} in[n']$ $\quad in[n] := gen[n] \cup (out[n] - kill[n])$

**Reaching Definitions** (forward, may) *

$in[n] := \bigcup_{n' \in pred[n]} out[n']$ $\quad out[n] := gen[n] \cup (in[n] - kill[n])$

**Available Expressions** (forward, must) *

$in[n] := \bigcap_{n' \in pred[n]} out[n']$ $\quad out[n] := gen[n] \cup (in[n] - kill[n])$

**Very busy Expressions** (backward, may) *

**Generic Iterative (Forward) Analysis**

for all n: $in[n] = \top$ $\quad out[n] := \top$

repeat for all n until no change:

$\quad in[n] := \sqcap out[n']$ $\quad n' \in pred[n]$

$\quad out[n] := F_n(in[n])$

**Meet operator** $\sqcap$: greatest lower bound

**Join operator** $\sqcup$: Least upper bound

* These are all distributive

**Loops**: A loop is a set of nodes in the CFG, with one distinguished entry point called the header.
- Every node is reachable from header & header is reachable from every node
- Nodes with outgoing edges are called loop exit nodes.

## Domination

- Node A dominates B if the only way to reach B from start is through A.
- **Back Edge** if target node (of back node) dominates the source node
- Domination is transitive & anti-symmetric (A dom. B & B dom. A → A = B)

**Dominator Dataflow Analysis**

$in[n] := \bigcap_{n' \in pred[n]} out[n']$

$out[n] := in[n] \cup \{n\}$

**Strictly dominates**: A sd B if A dominates B but also $A \neq B$

**Dominance Frontier** of node n:
1) Calc set of nodes it dominates
2) Calculate succ of those nodes (K)
3) Remove strict dominations from K

→ This is DF[n]

**Phi Placement (Easy Version)**

Place Nodes „maximally" (i.e. at every node with >= predecessors

# Garbage Collection

**Garbage:** X is reachable iff
- A register contains a pointer to X or
- Another reachable object y contains a pointer to X

↳ Unreachable objects are garbage

## Mark and Sweep

- Mark Phase: traces reachable objects
  - ↳ Has mark bit to use in mark phase
- Sweep phase: collects garbage objects
  - ⇒ Can fragment the memory
  - ⇒ Advantage: objects are not moved during GC

## Stop and Copy

Memory is organized into two areas
- Old space: used for allocation
- New space: used as reserve for GC
- Heap pointer points to the next free word in the old space

Garbage Collection:
- Copy all reachable objects from old into new
- After copy, the roles of old and new are reversed and the program resumes

⇒ When copying the elements, need to store forwarding pointer to the new copy (when we reach such an object, we know it's already copied)

⇒ Partition new space into 3 regions:

| copied & scanned | copied | empty |
|---|---|---|
| ↑ start    1) | ↑ scan    2) | ↑ alloc |

1) Objects whose pointer fields were scanned & fixed
2) Objects whose pointer fields weren't scanned

⇒ Must copy objects around → expensive

## Conservative Garbage Collection

- If a memory word looks like a pointer it is considered a pointer
- All such „pointers" are followed and we overestimate the reachable objects

## Reference Counting

- Store number of pointers to each object
- Each assignment operation has to manipulate the reference count
  - \+ Easy to implement
  - \+ Collects garbage incrementally without large pauses in the execution
  - − Manip. ref. counts at each assignment is slow
  - − Cannot collect circular structures