Queens' College Cambridge

Compiler Construction

Alistair O'Brien

Department of Computer Science

June 14, 2021

Contents

1 Lexical Analysis

- Lexical analysis (or lexing) is the first stage in the *frontend* of the compiler. Converts a sequence of characters or symbols into a sequence of *tokens*.

1.1 Regular Expressions and Languages

Definition 1.1.1. (**Regular expressions**) The set of regular expressions $\mathcal{R}_\Sigma \subseteq (\Sigma \cup \Omega)*$ over an alphabet Σ where $\Omega = \{\varnothing, \epsilon, *, |, (,)\}$ is defined by

$$\begin{aligned} r \ ::= \ & a \in \Sigma \ \mid \ \epsilon \ \mid \ \varnothing \\ \mid \ & (r_1) \mid (r_2) \\ \mid \ & (r_1)(r_2) \\ \mid \ & (r)^* \end{aligned}$$

- Precedence: $|$ < concatentation < $*$.
- $\equiv_{\mathcal{R}_\Sigma}$: $\mathcal{R}_\Sigma \longleftrightarrow \mathcal{R}_\Sigma$ is syntactic equivalence defined by ASTs of $\mathcal{R}_\Sigma$

Definition 1.1.2. (**Regular Language**) The regular language of the expression $r \in \mathcal{R}_\Sigma$, denoted $L(r) \subseteq \Sigma^*$, is inductively defined by

$$\begin{aligned} L(\varnothing) &= \emptyset \\ L(\epsilon) &= \{\varepsilon\} \\ L(a) &= \{a\} \\ L(r_1 \mid r_2) &= L(r_1) \cup L(r_2) \\ L(r_1 r_2) &= \{u_1 u_2 \in \Sigma^* : u_1 \in L(r_1) \wedge u_2 \in L(r_2)\} \\ L(r^*) &= \{u^n \in \Sigma^* : u \in L(r) \wedge n \in \mathbb{N}\} \end{aligned}$$

1.1.1 Finite Automata

Definition 1.1.3. (**NFA with ε-transitions**) A non-deterministic finite automaton (NFA) with ε-transition (NFA$^\varepsilon$) is a 5-tuple $M = (Q, \Sigma, \Delta_\varepsilon, q_0, A)$ where

(i) $Q = \{q_0, \ldots, q_n\}$ a finite set of states.

(ii) Σ is the finite alphabet of accepted input symbols.

(iii) $\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ the transition function.

(iv) $q_0 \in Q$ is the unique start (initial) state.

(v) $A \subseteq Q$ is the set of accepting states.

- $q \xrightarrow{x} q' \iff q' \in \Delta_\epsilon(q, x)$ where $x \in \Sigma \cup \{\varepsilon\}$.
- $q_0 \xrightarrow{u}^* q'$ to denote the transition path
$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q',$$
where $u = a_1 \ldots a_n \in \Sigma^*$.
- $q \overset{\varepsilon}{\Longrightarrow} q'$ is the reflexitive transitive closure of $\xrightarrow{\varepsilon}$.

Definition 1.1.4. (**ε-closure**) The ε-closure is defined the set of states reachable by zero or more ε-transitions:

$$\mathcal{E}(q) = \left\{ q' \in Q : q \overset{\varepsilon}{\Longrightarrow} q' \right\}.$$

- Algorithm for computing ε-closure:

```
E(q0) {
    S ← {q0}, l ← {};
    while (size(S) ≠ 0) {
        q ← pop(S);
        for (q' ∈ Δ(q, ε)) {
            l ← l ∪ {q};
            push(S, q');
        }
    }
    return l;
}
```

- The transition function $\Delta_\varepsilon^* : Q \times \Sigma^* \to \mathcal{P}(Q)$ over Σ^* is defined as

$$\begin{aligned}\Delta_\varepsilon^*(q, \varepsilon) &= \mathcal{E}(q) \\ \Delta_\varepsilon^*(q, ua) &= \bigcup_{q' \in \Delta_\varepsilon^*(q,u)} \left\{q'' \in Q : q' \overset{a}{\Longrightarrow} q''\right\}\end{aligned}$$

- $q \overset{a}{\Longrightarrow} q'$ consisting of the state transition sequence

$$q \overset{\varepsilon}{\Longrightarrow} \cdot \overset{a}{\to} \cdot \overset{\varepsilon}{\Longrightarrow} q'.$$

- The language accepted by the NFA$^\varepsilon$ is

$$L(M) = \left\{u \in \Sigma^* : q \overset{u}{\Longrightarrow} q' \in A\right\}.$$

Definition 1.1.5. (**Deterministic finite automaton**) A deterministic finite automaton (DFA) is a NFA $M = (Q, \Sigma, \Delta, q_0, A)$, with the property

$$\forall q \in Q, a \in \Sigma. \exists! q'. q \overset{a}{\to} q'.$$

- $\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ reduces to $\delta : Q \times \Sigma \to Q$.

- We define the transition function $\delta^* : Q \times \Sigma^* \to Q$ over Σ^* recursively as

$$\begin{aligned}\delta^*(q, \varepsilon) &= q \\ \delta^*(q, ua) &= \delta(\delta^*(q, u), a)\end{aligned}$$

- The language accepted by a DFA is

$$L(M) = \left\{u \in \Sigma^* : q_0 \overset{u}{\to}{}^* q' \in A\right\} = \{u \in \Sigma^* : \delta^*(q_0, u) \in A\}.$$

Theorem 1.1.1. (**Subset Construction**) For all NFA$^\varepsilon$ $M = (Q, \Sigma, \Delta_\varepsilon, q_0, A)$, there is a DFA $PM = (\mathcal{P}(Q), \Sigma, \delta, q_0', A')$ such that $L(M) = L(PM)$ where

(i) $\mathcal{P}(Q) = \{S : S \subseteq Q\}$

(ii) The transition function $\delta : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$

$$\delta(S, a) = \bigcup_{q \in S} \left\{q' \in Q : q \overset{a}{\Longrightarrow} q'\right\}.$$

(iii) The initial state is

$$q_0' = \left\{ q \in Q : q_0 \overset{\varepsilon}{\Longrightarrow} q \right\}.$$

(iv) The accepting states are

$$A' = \{S \in \mathcal{P}(Q) : S \cap A \neq \emptyset\}.$$

Proof. See IA Discrete Mathematics Notes. □

- Complexity: DFA of 2^n states $\implies$ $\Theta(2^n)$ worst-case time complexity for conversion.

Theorem 1.1.2. (**Kleene's Theorem**) A language L over Σ is regular $\iff$ it can be accepted by a DFA $M = (Q, \Sigma, \delta, q_0, A)$.

Proof. See IA Discrete Mathematics Notes. □

Corollary 1.1.2.1. We can construct an NFA$^\varepsilon M$ s.t $L(M) = L$, inductively: IMAGE

- **Generally**: We can construct an NFA$^\varepsilon$ for a given $r \in \mathcal{R}_\Sigma$, convert it into a DFA PM (using subset construction) and then use PM to determine whether $u \in \Sigma^*$ satisfies $u \in L(r)$.

- Complexity: $\Theta(|u|)$.

1.2 Lexical Grammars

- **Lexing Problem**: Given a finite set of tokens Λ, with associated regular expressions $\mathcal{R} = \{r_1, \ldots, r_n\} \subset \mathcal{R}_\Sigma$ and emitting function $T : \mathcal{R} \times \Sigma^* \to \Lambda$

  For the input $w \in \Sigma^*$, determine (u_i, Λ_{j_i}) for all $1 \leq i \leq m$ such that

  $$w = u_1 \ldots u_m \quad \text{and} \quad \forall 1 \leq i \leq m. u_i \in L(r_{j_i}).$$

- Resolving ambiguity:

  - Prioritize the order of regular expressions

- Use the longest matching regular expression to produce tokens.

Definition 1.2.1. (**Lexer**) A lexer L is the tuple $(Q, \Sigma, \Lambda, q_0, A, \delta, T)$, where $M = (Q, \Sigma, q_0, A\delta)$ is a DFA, $\{\text{error}\} \subseteq \Lambda$ is a finite set of *tokens*, and $T_L : A \times \Sigma^* \to \Lambda$ is a token emitting function.

- Lexer construction:
  1. For ordered set of regular expressions $\mathcal{R} = \{r_1, \ldots, r_n\}$ (1 highest priority, n lowest)
  2. Construct a NFA$^\varepsilon$ $M(r_i)$ for all $r_i \in R$. Construct $M = (Q, \Sigma, \Delta, q_0, \biguplus_i A_i)$ where $Q = q_0 \uplus \biguplus_i Q_i$.
  3. Use subset construction to build DFA PM. Each state $S \in A'$ is related to a $q_A^i = \max S$ (defined by highest priorities), an accepting state for r_i. Define emitting function:

     $$T_L(S, u) = T(r_i, u)$$

     where $q_A^i = \max S$.
- Lexer operation:
  1. Define set of dead states of lexer L

     $$D(L) = \left\{ q \in Q : \nexists q' \in A . u \in \Sigma^* . q \xrightarrow{u}{}^{*} q' \right\}.$$

  2. Transition until a dead state is reached. Then emit the token λ_i associated w/ the last accepting state q_A^i.
  3. Reset to start state.
- Lexers resolve ambiguity:
  - Each accepting state is associated with the *highest priority token*
  - Longest matches are determined using dead state transitions and last accepting states.

2 Syntactic Analysis

2.1 Context-Free Grammars

Definition 2.1.1. (**Context-Free Grammar**) A context free grammar (CFG) $G = (N, \Sigma, P, S)$ where

(i) N is a set of *non-terminal symbols*.

(ii) Σ is the alphabet, or set of *terminal symbols*.

(iii) $S \in N$ is the *start symbol*.

(iv) $P : N \to \mathcal{P}((N \cup \Sigma)^*)$ is the finite set of production rules, where $\{\alpha_1, \ldots, \alpha_n\} = P(A)$ is denoted

$$\begin{aligned} A &\longrightarrow \alpha_1 \\ &\vdots \\ A &\longrightarrow \alpha_n \end{aligned} \qquad \text{or} \qquad A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

- **Notation**: $\alpha, \beta, \gamma, \ldots \in (N \cup \Sigma)^*$, $X, Y, Z \in N \cup \Sigma$ and $A, B, C, \ldots \in N$.

Definition 2.1.2. (**Derivation**) For a grammar $G = (N, \Sigma, P, S)$, the relation $\underset{G}{\Longrightarrow}: (N \cup \Sigma)^* \longleftrightarrow (N \cup \Sigma)^*$ is

$$\alpha A \beta \underset{G}{\Longrightarrow} \alpha \gamma \beta \iff A \to \gamma \in P,$$

where $\alpha, \beta \in (N \cup \Sigma)^*$ and $A \in N$.

- **Notation**: $\underset{G}{\Longrightarrow}^*$ is the reflexive transitive closure of $\underset{G}{\Longrightarrow}$. The n-step composition of $\Longrightarrow$ is denoted $\Longrightarrow^n$.

- **Problem**: Derivation steps are *non-deterministic*
- **Solution**: Deterministic leftmost and rightmost derivations.

Definition 2.1.3. (**Leftmost and Rightmost Derivations**) A leftmost derivation is the relation $\overset{lm}{\underset{G}{\Longrightarrow}} : (N \cup \Sigma)^* \longrightarrow\!\!\!\!\!\!+\;\; (N \cup \Sigma)^*$ is

$$uA\alpha \overset{lm}{\underset{G}{\Longrightarrow}} u\beta\alpha \iff A \to \beta \in P,$$

where $u \in \Sigma^*, \alpha \in (N \cup \Sigma)^*$ and $A \in N$. Similarly, a rightmost derivation is the relation $\overset{rm}{\underset{G}{\Longrightarrow}} : (N \cup \Sigma)^* \longrightarrow\!\!\!\!\!\!+\;\; (N \cup \Sigma)^*$ is

$$\alpha Au \overset{rm}{\underset{G}{\Longrightarrow}} \alpha\beta u \iff A \to \beta \in P.$$

Definition 2.1.4. (**Context-Free Language**) The language generated by the grammar $G = (N, \Sigma, P, S)$ is

$$L(G) = \left\{ u \in \Sigma^* : S \underset{G}{\Longrightarrow}^* u \right\}.$$

A formal language $L \subseteq \Sigma^*$ is a *context-free language* (CFL) $\iff L = L(G)$ for some grammar G.

Lemma 2.1.1. (**Equivalence of Derivations**) For all grammars $G = (N, \Sigma, P, S)$,

$$\begin{aligned} \forall u \in \Sigma^*. S \underset{G}{\Longrightarrow} u &\iff S \overset{lm}{\underset{G}{\Longrightarrow}} u \\ &\iff S \overset{rm}{\underset{G}{\Longrightarrow}} u \end{aligned}$$

- **Idea**: Proofs (and algorithms) may now use leftmost or rightmost derivations without loss of generality $\implies$ LL(k) and LR(k)

2.1.1 Concrete Syntax Trees

Definition 2.1.5. (**Concrete Syntax Trees**) For a grammar $G = (N, \Sigma, P, S)$, the set of concrete syntax trees (CSTs) $\mathscr{T}(G)$ is inductively defined as

$$\text{(Sym)}\ \frac{\alpha \in \Sigma \cup N}{(\alpha, \langle\rangle)} \qquad (\varepsilon)\ \frac{A \to \varepsilon \in P}{(A, \langle(\varepsilon, \langle\rangle)\rangle)}$$

$$\text{(Prod)}\ \frac{\overbrace{(\alpha_1, \ldots)}^{t_1} \quad \cdots \quad \overbrace{(\alpha_n, \ldots)}^{t_n} \quad A \to \alpha_1 \ldots \alpha_n \in P}{(A, \langle t_1, \ldots, t_n\rangle)}$$

where $(x, \langle t_1, \ldots, t_n\rangle)$ denotes a tree with root x and children $t_1, \ldots, t_n$.

- CSTs are inductively associated with derivations.

Definition 2.1.6. (**Frontier**) The *frontier* of a concrete syntax tree $t \in \mathscr{T}(G)$, defined by $\mathcal{F} : \mathscr{T}(G) \to (N \cup \Sigma)^*$:

$$\begin{aligned} \mathcal{F}\left((\alpha, \langle\rangle)\right) &= \alpha \\ \mathcal{F}\left((A, \langle t_1, \ldots, t_n\rangle)\right) &= \mathcal{F}(t_1) \ldots \mathcal{F}(t_n) \end{aligned}$$

for $\alpha \in \Sigma \cup N \cup \{\varepsilon\}$

- A CST is said to be *complete* or a *parse tree* iff $\mathcal{F}(t) \in \Sigma^*$.

Theorem 2.1.1. For a grammar $G = (N, \Sigma, P, S)$,

$$\forall A \in N, \alpha \in (N \cup \Sigma)^*. A \underset{G}{\Longrightarrow}^* \alpha \implies \exists t \in \mathscr{T}(G).\text{Root}(t) = A \wedge \mathcal{F}(t) = \alpha.$$

2.1.2 Regular Languages

Theorem 2.1.2. A language L is regular $\iff$ $L = L(G)$ for a grammar $G = (N, \Sigma, P, S)$ where all production rules are of the form

$$\begin{aligned} A &\longrightarrow xB \\ A &\longrightarrow x \\ A &\longrightarrow \varepsilon \end{aligned}$$

Proof. By Kleene's theorem, there exists a DFA $M = (Q, \Sigma, \delta, q_0, A)$ such that $L = L(M)$.

We introduce the following grammar $G = (Q, \Sigma, P, q_0)$, where P is defined by

$$q \to aq' \iff q \xrightarrow{a} q' \qquad q \to \varepsilon \iff q \in A$$

We proceed by induction over Σ^*, with the statement

$$P(u) = q_0 \xrightarrow{u}^* q' \iff q_0 \Longrightarrow uq'.$$

□

2.1.3 Closures of Context-Free Languages

- Regular languages are closed under $\cup, \cap$, concatentation, complementation and $*$.

Theorem 2.1.3. Let $\mathcal{L}_\Sigma^{CFG}$ be the set of context free languages.

$$\forall L_1, L_2 \subseteq \Sigma^*. L_1, L_2 \in \mathcal{L}_\Sigma^{CFG} \implies L_1 \cup L_2, L_1 L_2, L_1^* \in \mathcal{L}_\Sigma^{CFG}$$

- Unlike regular languages, CFLs aren't closed under $\cap$ and complementation.

Theorem 2.1.4. Context free languages are not closed under intersection, that is

$$\exists L_1, L_2 \subseteq \Sigma^*. L_1, L_2 \in \mathcal{L}_\Sigma^{CFG} \implies L_1 \cap L_2 \notin \mathcal{L}_\Sigma^{CFG}.$$

Proof. (*Sketch*)
We introduce the witnesses

$$L_1 = \left\{a^i b^j c^k \in \Sigma^* : i < j\right\}$$
$$L_2 = \left\{a^i b^j c^k \in \Sigma^* : i < k\right\}$$

where $\Sigma = \{a, b, c\}$. L_1 and L_2 are context-free. However,

$$L_1 \cap L_2 = \left\{a^i b^j c^k \in \Sigma^* : i < j \wedge i < k\right\},$$

is not context-free. □

Theorem 2.1.5. Context free languages are not closed under complementation, that is

$$\exists L \subseteq \Sigma^*. L \in \mathcal{L}_\Sigma^{CFG} \implies L^c \notin \mathcal{L}_\Sigma^{CFG}.$$

Proof. We proceed by contradiction. Let us assume that

$$\forall L \subseteq \Sigma^*. L \in \mathcal{L}_\Sigma^{CFG} \implies L^c \in \mathcal{L}_\Sigma^{CFG}.$$

Let $L_1, L_2 \in \mathcal{L}_\Sigma^{CFG}$ be arbitrary CFLs. So we have $L_1^c, L_2^c \in \mathcal{L}_\Sigma^{CFG}$. By theorem ??, we have $L_1^c \cup L_2^c \in \mathcal{L}_\Sigma^{CFG}$. By De Morgan's law (for sets), we have

$$(L_1^c \cup L_2^c)^c = L_1 \cap L_2 \in \mathcal{L}_\Sigma^{CFG}.$$

Since L_1, L_2 are arbitrary, this contradicts theorem ??. So we are done. □

2.1.4 Decision Problems

Theorem 2.1.6. For $u \in \Sigma^*$ and grammar G, determining $u \in L(G)$ is decidable.

Proof. PDAs or Early's Parsing Algorithm □

Theorem 2.1.7. For a grammar G, determining $L(G) = \emptyset$ is decidable.

Proof. (*Sketch*)
For a grammar $G = (N, \Sigma, P, S)$, we may compute a string $u \in \Sigma^*$ by considering concrete syntax trees of height $\leq |N|$ since a grammar should be able to generate a string without recursion. □

- Many problems for CFLs are **undecidable**:
  - For CFLs $L_1, L_2 \in \mathcal{L}_\Sigma^{CFG}$, determine $L_1 = L_2$
  - For CFLs $L_1, L_2 \in \mathcal{L}_\Sigma^{CFG}$, determine $L_1 \subset L_2$
  - For CFG G, determine whether G is ambiguous

2.2 Push-Down Automata

- **Problem**: By theorem ?? and the *Pumping lemma*, CFGs are unrecognizable by *finite* automata.
- **Solution**: Adding memory (a stack) to an NFA$^\varepsilon$ $\implies$ PDA

Definition 2.2.1. (**Push-Down Automaton**) A push-down automaton (PDA) is a 6-tuple $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, A)$ where:

(i) Q is a finite set of states

(ii) Σ is an alphabet of input symbols

(iii) Γ is an alphabet of stack symbols

(iv) $\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is a non-deterministic transition function

(v) $q_0 \in Q$ is the initial state

(vi) $Z \in \Gamma$ is the initial stack symbol

(vii) $A \subseteq Q$ is the set of accepting states

Definition 2.2.2. (**Configuration**) For a PDA $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, A)$, the tuple $(q, u, S) \in Q \times \Sigma^* \times \Gamma^*$ is a *configuration*, denoting the current state q, the remaining input u and S is the stack.

The set of configurations of M is denoted $\mathscr{C}(M)$.

- A transition $(q', S') \in \Delta(q, a, \alpha)$ denotes M stepping from the state q with current symbol a and top of stack α to the state q', popping α and pushing S'.

Definition 2.2.3. (**Transition Relation**) The transition relation $\longrightarrow_M$: $\mathscr{C}(M) \nrightarrow \mathscr{C}(M)$ defined by

$$\frac{(q', S') \in \Delta(q, a, \alpha)}{(q, au, \alpha S) \longrightarrow_M (q', u, S'S)}$$

$$\frac{(q', S') \in \Delta(q, \varepsilon, \alpha)}{(q, u, \alpha S) \longrightarrow_M (q', u, S'S)}$$

- $\longrightarrow^*$ denotes the reflexive transitive closure of $\longrightarrow$.

Definition 2.2.4. (**Language**) The language accepted by a PDA $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, A)$, denoted $L(M)$, is defined by

$$L(M) = \{u \in \Sigma^* : \exists q \in A.(q_0, u, Z) \longrightarrow^* (q, \varepsilon, \varepsilon)\}\,.$$

2.2.1 Kleene's Theorem II

Theorem 2.2.1. (**Kleene's Theorem II**) A language L over Σ is context free $\iff$ it can be accepted by a PDA $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, A)$.

Proof.
($\implies$). Let $G = (N, \Sigma, P, S)$ be a grammar. We wish exhibit a PDA M s.t $L(M) = L(G)$.

Let $M = (\{q_0, q\}, \Sigma, \Gamma, \Delta, q_0, Z, \{q\})$, where $\Gamma = \Sigma \cup N \cup \{Z\}$ and Δ is defined as follows:

$$\begin{aligned}\Delta(q_0, \varepsilon, Z) &= \{(q, S)\} \\ \Delta(q, \varepsilon, A) &= \bigcup \{(q, \alpha) : A \to \alpha \in P\} \\ \Delta(q, a, a) &= \{(q_0, \varepsilon)\}\end{aligned}$$

for all $a \in \Sigma$ and $A \in N$. We wish to show that $L(M) = L(G)$, by theorem ??, that is

$$S \overset{lm}{\Longrightarrow}{}^{*} u\alpha \iff \forall v \in \Sigma^*.(q, uv, S) \longrightarrow_M^* (q, v, \alpha),$$

where $\alpha \in N(N \cup \Sigma)^* \cup \{\varepsilon\}$ and $u \in \Sigma^*$. By lemma ??, we are done.
($\impliedby$). Let $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, A)$ be an arbitrary PDA. Let $q' \in A$ be an arbitrary accepting state. Instantiating lemma ?? with $q = q_0$ and $\alpha = Z$, we have $L(M) = L(G^{\alpha}_{q,q'}) = L$

$\square$

Lemma 2.2.1. For G and M as defined in the proof of theorem ??,

$$S \overset{lm}{\Longrightarrow}{}^{*} u\alpha \iff \forall v \in \Sigma^*.(q, uv, S) \longrightarrow_M^* (q, v, \alpha),$$

where $\alpha \in N(N \cup \Sigma)^* \cup \{\varepsilon\}$ and $u \in \Sigma^*$.

Lemma 2.2.2. For any PDA $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, A)$ and for all $S \subseteq Q, (q, q') \in Q \times Q, \alpha \in \Gamma$, there exists a grammar $G^{S,\alpha}_{q,q'}$ s.t

$$L(G^{S,\alpha}_{q,q'}) = \{u \in \Sigma^* : (q, u, \alpha) \longrightarrow_M^* (q', \varepsilon, \varepsilon)\},$$

and S is the set of intermediate states in the transition sequence.

Proof. (*Sketch*) Let $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, A)$ be an arbitrary PDA. We introduce the grammar $G^{S,\alpha}_{q,q'} = ((S \cup \{q, q'\}) \times \Gamma, \Sigma, P, (q, q', \alpha))$, where P is defined by

$$\begin{array}{lll} (q^0, q^j, \beta) \longrightarrow a(q^1, q^2, \beta_1)(q^2, q^3, \beta_2) \dots (q^j, q^j, \beta_j) & \iff & (q^1, \beta_1 \dots \beta_n) \in \Delta(q^0, a, \beta) \\ (q^0, q^1, \beta) \longrightarrow a & \iff & (q^1, \varepsilon) \in \Delta(q^0, a, \beta) \end{array}$$

for all $(q^i, q^j, \beta) \in N$. We deduce that $L(G^{S,\alpha}_{q,q'}) = \{u \in \Sigma^* : (q, u, \alpha) \longrightarrow^*_M (q', \varepsilon, \varepsilon)\}$.

Formally, we'd proceed by induction on $|S|$. □

2.2.2 Deterministic PDAs

Definition 2.2.5. (**Deterministic PDAs**) A PDA $M = (Q, \Sigma, \Gamma, \Delta, q_0, Z, A)$ is said to be *deterministic* if

$$\Delta : Q \times \Sigma \times \Gamma \to Q \times \Gamma^*.$$

- Any CFG G that has a equivalent deterministic PDA is a *deterministic* GFG.
- **Problem**: Not all CFGs are deterministic e.g.

$$\left\{x^k y^k z^k : k \geq 0\right\}.$$

2.3 Grammar Transformations

- Parsers are designed to be *deterministic*
- **Problem**: Choosing a production rule $A \to \gamma$ for a non-terminal A is *non-deterministic*.
- **Solution**: Parsers make decisions based on the *next symbols* (the lookahead). Grammars are *transformed* for parsing.

2.3.1 Left Factoring

- **Problem**: For a production rule $A \to \alpha\beta \mid \alpha\gamma$, given the lookhead α, cannot determine which expansion of A to use for the derivation.

- **Solution**: Left-factoring.

Definition 2.3.1. The grammar transformation *left-factoring* of a grammar $G = (N, \Sigma, P, S)$ with production rule

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m \in P,$$

is a grammar G_{lf} defined by the transformation:

$$\frac{A \to \alpha\beta_1 \mid \cdots \alpha\beta_n \mid \gamma_1 \mid \cdots \mid \gamma_m}{A \to \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_m, \quad A' \to \beta_1 \mid \cdots \mid \beta_n}$$

where $\alpha \neq \varepsilon$

Theorem 2.3.1. For all grammars $G = (N, \Sigma, P, S)$, $L(G) = L(G_{lf})$

- Useful for *top-down parsing.*

2.3.2 Left Recursion

Definition 2.3.2. (**Left Recursive**) A grammar $G = (N, \Sigma, P, S)$ is (immediately) *left recursive* iff

$$A \to A\alpha_1 \mid \cdots A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m \in P,$$

where $\alpha_i, \beta_j \in (N \cup \Sigma)^*$ and $\beta_j \neq A\gamma$ for some $\gamma \in (N \cup \Sigma)^*$.

- **Problem**: Left recursion results in infinite loops for a top-down parser applying the rule $A \to A\alpha_i$.

Definition 2.3.3. (**Eliminating Left Recursion**) The grammar transformation *eliminating (immediate) left recursion* of a (immediate) left recursive grammar G is a grammar G_{lr} defined by the transformation

$$\frac{A \to A\alpha_1 \mid \cdots A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m}{A \to \alpha\beta_1 A' \mid \cdots \mid \beta_m A', \quad A' \to \alpha_1 A' \mid \cdots \mid \alpha_n A' \mid \varepsilon}$$

Theorem 2.3.2. For all grammars $G = (N, \Sigma, P, S)$, $L(G) = L(G_{lr})$

- **Problem**: Transformation doesn't eliminate *indirect* left recursion

Definition 2.3.4. (**(Indirectly) Left Recursive**) A grammar $G = (N, \Sigma, P, S)$ is (indirectly) *left recursive* iff

$$\begin{aligned} A_0 &\longrightarrow A_1\alpha_1 \mid \cdots \\ A_1 &\longrightarrow A_2\alpha_2 \mid \cdots \\ &\vdots \\ A_n &\longrightarrow A_0\alpha_{n+1} \mid \cdots \end{aligned}$$

are production rules in P.

- See notes for indirect left recursion elimination.

2.3.3 Ambiguity

Definition 2.3.5. (**Ambiguity**) A grammar $G = (N, \Sigma, P, S)$ is *ambiguous* if there exists $u \in \Sigma^*$ s.t there exists two distinct derivations of u.

IMAGE

- **Problem**: Ambiguity effects the *time complexity* of parsers.
- Ambiguity is removed by adding additional rules. e.g. rules that denote *precedence* or *associativity*.
- EXAMPLE
- **Problem**: Some grammars are *inherently* ambiguous e.g.

$$L = \{a^m b^m c^n : m, n \geq 1\} \cup \{a^m b^n c^n : m, n \geq 1\}.$$

2.4 Top-Down Parsing

Definition 2.4.1. (**Top-Down Parsing**) Top down parsing is the problem of determining $S \underset{G}{\Longrightarrow}{}^*_{lm} u$ given G and $u \in \Sigma^*$.

- A top-down parser is an algorithm that decides top-down parsing.

2.4.1 Recursive Descent Parsing

- *Recursive-descent parser*: For grammar $G = (N, \Sigma, P, S)$, a recursive-decent parser consists of a set of functions $\{p_A : A \in N\}$ implements the non-terminal rule $A \in N$. e.g.

```
void p_A() {
    // A → α_1 ... α_k
    for (int i = 1; i < k; i++) {
        if (α_i ∈ N) p_{α_i}();
        else if (α_i ∈ Σ && α_i = peek()) next();
        else throw ParserError();
    }
}
```

- Backtracking is typically implemented. Suitable for goal-directed languages e.g. Prolog.
- **Problem**: Recursive decent parsers cannot parse left-recursive grammars, since for $A \to A\alpha$, $p_A()$ would infinitely call $p_A()$.
- **Solution**: Left factoring. See ??

2.4.2 First and Follow

- **Idea**: Parsing is equivalent to determining which production rule to "expand" at any stage during a $\underset{G}{\Longrightarrow}$ derivation.
- **Problem**: Approaches using backtracking $\Longrightarrow$ inefficiency and complexity.
- **Solution**: The *notion* of lookahead.

Definition 2.4.2. (**Lookahead**) A *lookahead* of k is the ability to analyze the k next symbols (and the current symbol) to determine production rules that may be applied.

- Consider the input $u = v\mathbf{a}w$, $v, w \in \Sigma^*$ with current symbol $\mathbf{a} \in \Sigma^k$. Suppose we have the derivation $S \underset{lm}{\Longrightarrow} vA\beta$.

- A parser must determine the derivation $A\beta \underset{lm}{\Longrightarrow} \mathbf{a}w$. It considers cases of the production rule of A:
  1. If $A \longrightarrow \mathbf{b}\gamma$, then $\mathbf{a} = \mathbf{b}$ for a derivation to exist.
  2. If $A \longrightarrow B\gamma$, then
     (a) Determine whether $B \underset{lm}{\Longrightarrow} \mathbf{a}\delta$.
     (b) If $B \underset{lm}{\Longrightarrow} \varepsilon$, then determine whether $\delta \underset{lm}{\Longrightarrow} \mathbf{a}\delta$
  3. If $A \longrightarrow \varepsilon$, then determine whether $\beta \underset{lm}{\Longrightarrow} \mathbf{a}\delta$
- **Idea**: This notion of checking cases is formalized via First and Follow.

Definition 2.4.3. (**First**) The function $\mathsf{First} : (N \cup \Sigma)^* \to \mathcal{P}(\Sigma)$ is defined inductively over $(N \cup \Sigma)^*$ as

$$\begin{aligned}
\mathsf{First}(a) &= \{a\} && a \in \Sigma \\
\mathsf{First}(\varepsilon) &= \{\varepsilon\} \\
\mathsf{First}(A) &= \bigcup_{i=1}^{n} \mathsf{First}(\alpha_i) && A \to \alpha_1 \mid \cdots \mid \alpha_n \in P \\
\mathsf{First}(X_1 \dots X_n) &= \bigcup_{i=1}^{k} \mathsf{First}(X_i) && k = \max\left\{k : \bigwedge_{i=1}^{k} X_i \Longrightarrow^* \varepsilon\right\}
\end{aligned}$$

or

$$\mathsf{First}(\alpha) = \{a \in \Sigma : \exists \beta \in (N \cup \Sigma)^*. \alpha \Longrightarrow^* a\beta\}$$

- $\mathsf{First}(\alpha)$ is the set of terminals that prefix sentences derived from α.
- $\mathsf{First}(A)$ is used for cases (1) and (2.a)

Theorem 2.4.1.

$$\alpha \underset{lm}{\Longrightarrow} \varepsilon \implies \varepsilon \in \mathsf{First}(\alpha),$$
$$\forall i \in \mathbb{Z}^+. \left(X_1 \dots X_i \underset{lm}{\Longrightarrow} \varepsilon\right) \implies \mathsf{First}(X_{i+1}) \subseteq \mathsf{First}(X_1 \dots X_n),$$
$$\varepsilon \in \mathsf{First}(X_1 \dots X_n) \implies \forall i \in \mathbb{Z}^+. \varepsilon \in \mathsf{First}(X_i).$$

- Follow is used when the production rule is *nullable* (derives ε).

Definition 2.4.4. (**Follow**) The function $\mathsf{Follow} : N \to \Sigma$ for the grammar $G = (N, \Sigma, P, S)$ where $S \to \alpha\$ \in P$ and $\$ \in \Sigma$ is the *EOF* token, is given by

$$\begin{aligned}
\mathsf{Follow}(S) &= \{\$\} \\
\mathsf{Follow}(B) &\supseteq \mathsf{First}(\beta) \setminus \{\varepsilon\} && \text{if } A \to \alpha B \beta \\
\mathsf{Follow}(B) &\supseteq \mathsf{Follow}(A) && \text{if } A \to \alpha B \beta \wedge \varepsilon \in \mathsf{First}(\beta)
\end{aligned}$$

or

$$\mathsf{Follow}(A) = \{a \in \Sigma : \exists \alpha, \beta \in (N \cup \Sigma)^*. S \Longrightarrow \alpha A a \beta\}$$

- Follow returns the set of terminals that immediately follow a non-terminal.
- $\mathsf{Follow}(A)$ is used for cases (2.b) and (3).
- First and Follow may be extended for a k lookahead.

2.4.3 Predictive Parsers and $LL(k)$

- **Idea**: Pre-compute table of decisions for derivation case analysis.

Definition 2.4.5. (**Parse Table**) We define a *parse table* $\mathbf{T}^G$ as a matrix (two-dimensional array) of order $|N| \times |\Sigma|$, where the entry $\mathbf{T}^G_{Aa}$ contains the set of production rules that may be "expanded" given current symbol a, that is

$$A \to \alpha \in \mathbf{T}^G_{Aa} \iff (a \in \mathsf{First}(\alpha)) \vee (\varepsilon \in \mathsf{First}(\alpha) \wedge a \in \mathsf{Follow}(A))$$

- With the parse table $\mathbf{T}^G$ of G constructed, we use a *stack* to keep track of the left-most non-terminal producing the following *predictive parsing* algorithm:

```
stack ← {S};
while ((X = peek(stack)) ≠ $) {
    a ← peek();

    if (X = a) {
```

```
            pop(stack);
            next();
        } else {
            non-deterministically select X → α from T^G_{Xa};
            pop(stack);
            // push α onto stack (w/ leftmost symbol on top)
            for (i = |α|; i > 0; i--) push(stack, α_i);
        }
    }
```

- **Problem**: Predictive parsers are (in general) *non-deterministic* $\implies$ consider the restricted set of grammars with determinism ($LL(k)$).

Definition 2.4.6. ($LL(k)$) The class of grammar, denoted $LL(k)$, is the set of grammars G whose parsing tables $\mathbf{T}^G$ contain a single production rule.

- $LL(k)$ denotes a left-to-right parser, using left-most derivations with a k symbol lookahead. e.g. $LL(0)$ is the set of *regular grammars*.
- $LL(k)$ have *deterministic* predictive parsing algorithms.
- **Problem**: Grammar transformations: eliminating *left-recursion* and *left-factoring* are *often* applied to make a language $LL(k)$, since left-factors and left-recursion results multiple entries in the parsing table.
- **Problem**: $k \geq 2$ results in larger parsing tables $\implies$ unacceptable memory usage.

2.5 Bottom-Up Parsing

- **Problem**: Top down parsing has unacceptable memory usage for $k \geq 2$ and additional grammar transformations.

Definition 2.5.1. (**Reduction**) $\alpha \in (N \cup \Sigma)^*$ reduces to $\beta \in (N \cup \Sigma)^*$, denoted $\alpha \underset{G}{\rightarrowtail} \beta$ in $G \iff \beta \underset{G}{\implies} \alpha$.

- **Idea**: Bottom up parsing: determine whether $u \underset{G}{\rightarrowtail} S$.

Definition 2.5.2. (**LR Configuration**) A configuration $c \in \mathscr{C}(G)$ for G is a pair (α, v), where u is parsed and reduces to α, and v is the remaining input. c is *valid* if $S \underset{G}{\Longrightarrow} \alpha v$.

Definition 2.5.3. (**LR Transitions**) The transition relation $\longrightarrow: \mathscr{C}(G) \nrightarrow \mathscr{C}(G)$ is defined by

(i) **Shift**: Shift the remaining input:

$$\frac{}{(\alpha, av) \overset{shift}{\longrightarrow} (\alpha a, v)}$$

(ii) **Reduce**: Apply reduction $\alpha \rightarrowtail \beta$:

$$\frac{}{(\alpha, v) \overset{reduce}{\longrightarrow} (\beta, v)}$$

- **Problem**: Non-determinism of $\longrightarrow$ transitions, on both *what to reduce* and *when to shift or reduce.*

- **Solution**: Use a *rightmost* derivation $\Longrightarrow_{rm} \implies \alpha$ is a *stack* w/ reductions applied to matching production rules:

$$\frac{A \to \beta \in P}{(\alpha\beta, v) \overset{reduce}{\longrightarrow} (\alpha A, v)}$$

- **Solution**: Restrict grammars for determinism $\implies LR(k)$

Definition 2.5.4. ($LR(k)$) $LR(k)$ is the class of grammars G with left-to-right parsers, using right-most derivations with a k symbol lookahead.

2.5.1 LR(0)

- LR(0) grammars have 0 symbol lookahead. More powerful than LL(0) (regular languages)

2.5.1.1 LR(0) Items

- **Problem**: Determining transitions on configurations
- **Solution**: *Valid* items determine possible transitions

Definition 2.5.5. (**LR(0) Item**) For a production rule $A \to \alpha\beta \in P$, for a grammar $G = (N, \Sigma, P, S)$,

$$A \to \alpha \cdot \beta,$$

denotes an *item* where $u \underset{G}{\rightarrowtail} \alpha$ (u has been shifted / *parsed*), with $\cdot$ denoting the current position, and $v \underset{G}{\rightarrowtail} \beta$ is *expected*. The set of LR(0) items of G is denoted $\mathscr{I}(G)$.

- $A \to \alpha \cdot \beta$ is an item corresponding to some substring uv w/ u having been parsed:

$$a_0 \ldots \underbrace{\overbrace{\overbrace{a_k \ldots a_{i-1}}^{\alpha} \cdot \overbrace{a_i \ldots}^{\beta}}^{A}}_{uv} \ldots a_{n-1}.$$

- An item $A \to \alpha \cdot \beta$ represents a *possible* reduction of A.
- **Note**:
  - $A \to \gamma \cdot \implies$ *reduce* A
  - $A \to \alpha \cdot B\beta \implies$ examine the rules $B \to \gamma \in P$.

Definition 2.5.6. (**Valid Item**) An item $A \to \beta \cdot \gamma$ is *valid* in $\alpha\beta \iff \exists v \in \Sigma^*.\alpha\beta\gamma v \underset{G}{\rightarrowtail} S$.

- Each configuration $c = (\alpha\beta, v)$ has a set valid items of the form: $A \to \beta \cdot \gamma$ in $\alpha\beta$.

Definition 2.5.7. (**Kernel and Non-Kernel Items**) *Kernel items* are items of the form: $A \to \alpha \cdot \beta$ where $\alpha \neq \varepsilon$. *Non-kernel items* are items of the form: $A \to \cdot\gamma$

- Examining $B \to \gamma \in P$ consists of *non-deterministically* transitioning to a non-kernel item of B.

- **Idea**: Items $\mathscr{I}(G)$ form states Q of an NFA$^\varepsilon$ $M'(G)$ w/ accepting states corresponding to reductions and transitions corresponding to shifts.

Definition 2.5.8. (**Augmented Grammar of** G) Let $G = (N, \Sigma, P, S)$ be a CFG. Define $G' = (N \cup \{S'\}, \Sigma \cup \{\$\}, P', S')$ where $P' = P \cup \{S' \to S\$\}$ and $\$ \notin \Sigma, S' \notin N$ be the *augmented grammar* of G.

- Augmented grammars are required for a correct initial state: $S' \to \cdot S\$$.

Definition 2.5.9. ($LR(0)$ **NFA**) Let G' be the augmented grammar of G. Define $M(G)$ be the NFA$^\varepsilon$ $M(G) = (\mathscr{I}(G'), N \cup \Sigma, \Delta_\varepsilon, q_0, A)$ where $q_0 = S' \to \cdot S\$$ and $A = \{A \to \alpha \cdot \in \mathscr{I}(G'), A \to \alpha \in P'\}$, and Δ_ε is defined by:

$$A \to \alpha \cdot X\beta \xrightarrow{X} A \to \alpha X \cdot \beta \qquad \text{Shift}$$
$$A \to \alpha \cdot B\beta \xrightarrow{\varepsilon} B \to \cdot\gamma \qquad \text{Examine}$$

- ε-transitions intuitionally relate to parsing "subgoals".

Theorem 2.5.1. ($LR(0)$ **Validity Theorem**) $A \to \beta \cdot \gamma \in \Delta_\varepsilon^*(q_0, \alpha\beta) \iff A \to \beta \cdot \gamma$ is valid in $\alpha\beta$ in G', the augmented grammar.

- **Consequence**: Validity may be used as a predicate for shift / reduce transitions on configurations $\mathscr{C}(G)$ *if* each config has a unique valid item. (*No conflicts*)

  $LR(0)$ configuration transitions:

  (i) **Shift**:

  $$\frac{A \to \beta \cdot a\gamma \text{ is valid in } \alpha\beta}{(\alpha, av) \xrightarrow{shift} (\alpha a, v)}$$

  (ii) **Reduce**:

  $$\frac{A \to \beta \cdot \text{ is valid in } \alpha\beta}{(\alpha\beta, v) \xrightarrow{reduce} (\alpha A, v)}$$

- **Problem**: $M(G)$ is *non-deterministic* $\implies$ Subset construction to produce DFA $PM(G)$

Definition 2.5.10. (**Item Closure**) The ε-closure of an LR(0) item $A \to \alpha \cdot \beta$, denoted $\mathcal{E}(A \to \alpha \cdot \beta)$, is the set of items inductively defined by

$$\frac{}{A \to \alpha \cdot \beta}$$

$$\frac{A \to \alpha \cdot B\beta \qquad B \to \gamma}{B \to \cdot\gamma}$$

- Intuitively: $\mathcal{E}(A \to \alpha \cdot \beta)$ is all the items we can "transition" to without *shifting*.
- $\mathcal{E}$ is the closure used for subset construction, with each $\mathcal{E}(A \to \alpha \cdot \beta)$ being a state in $PM(G)$.
- $LR(0)$ parser:

```
S ← {q₀} // State stack. Current state = TOS
c ← { sentinel_stack: {}, input: w } // Configuration

while (size(c.input) ≠ 0) {
    q, a ← peek(S), peek(c.input);

    // Non-deterministically decide:
    q' ← δ(q, a);
    if (A → α · aβ ∈ q') {
        push(c.sentinel_stack, a); pop(c.input); // Shift a
        push(S, q); // transition to q'
    }
    if (A → α· ∈ q') {
        // Pop α and |α| states
        for (i ← |α|; i > 0; i--) {pop(c.sentinel_stack, αᵢ); pop(S)}
        push(c.sentinel_stack, A);

        // "Shift" A:
        push(S, δ(peek(S), A));
    }
}

// No more input, check current state
if (S' → S$· ∈ peek(S)) return ACCEPT;
throw new ParserError();
```

- **Problem**: Possible for multiple items to match either (or both) `if` conditions. See section ??

- **Problem**: Poor performance using sets $\implies$ precompute actions in a parsing table, enumerating states (denoted q_i)

Definition 2.5.11. ($LR(0)$ **Parsing Table**) A $LR(0)$ parse table $\mathbf{T}^G$ is the pair of matrices ($\mathbf{A}^G$, $\mathbf{G}^G$)

$\mathbf{A}^G$ is a matrix of order $|Q| \times |\Sigma|$, where each entry $\mathbf{A}^G_{qa}$ contains the set of *actions* defined by

$$\texttt{action ::= shift } q \texttt{ | reduce } A \to \beta$$

and $\mathbf{G}^G$ is a matrix of order $|Q| \times |N|$, where each entry $\mathbf{G}^G_{qA}$ corresponds to a *goto* action, the "shift" transition following a reduction on A.

IMAGE

2.5.1.2 Inadequate States

- **Problem**: If state contains a reducible item ($A \to \alpha\cdot$) and has terminal transitions $\implies$ non-determinism. A *shift/reduce* conflict.

- *reduce/reduce* conflicts also occur when a state contains 2 distinct reducible items.

Definition 2.5.12. (**Inadequate State**) A LR(0) DFA containing a *shift/reduce* or *reduce/reduce* conflicts is said to have *inadequate states*. A DFA with inadequate states is said to be *inadequate*

Definition 2.5.13. ($LR(0)$) A grammar G is said to be $LR(0)$ if it's $LR(0)$ DFA $PM(G)$ is adequate.

- $LR(0)$ grammars have a $LR(0)$ parsing table $\mathbf{T}^G$ w/ each entry in $\mathbf{A}^G$ containing either a single shift action or a single reduce action.

2.5.2 SLR(1)

- **Problem**: Conflicts.

- **Solution**: Add a lookahead.

Definition 2.5.14. ($SLR(1)$) $SLR(1)$, or Simple $LR(1)$, is an extension of $LR(0)$ with a $k = 1$ lookahead.

- **Idea**: If we're in a state w/ item $A \to \beta\cdot$, then our configuration $c = (\alpha\beta, v)$ transitions to $c = (\alpha A, v)$. Corresponds to a reduction $\beta a \rightarrowtail Aa$, where $a \in \mathsf{Follow}(A)$.
- $SLR(1)$ configuration transitions:
  (i) **Shift**:
  $$\frac{A \to \beta \cdot a\gamma \text{ is valid in } \alpha\beta}{(\alpha, av) \xrightarrow{shift} (\alpha a, v)}$$
  (ii) **Reduce**:
  $$\frac{A \to \beta\cdot \text{ is valid in } \alpha\beta \qquad a \in \mathsf{Follow}(A)}{(\alpha\beta, av) \xrightarrow{reduce} (\alpha A, av)}$$
- Consequences:
  $$\begin{aligned} A \to \alpha \cdot a\beta \in q \wedge \delta(q,a) = q' &\implies \text{shift } q' \in \mathbf{A}^G_{qa} \\ A \to \alpha\cdot \in q \wedge A \neq S' &\implies \forall a \in \mathsf{Follow}(A).\text{reduce } A \to \alpha \in \mathbf{A}^G_{qa} \\ \delta(q, A) = q' &\implies \mathbf{G}^G_{qA} = q' \end{aligned}$$

  Problem: $SLR(1)$ parsing tables $\mathbf{T}^G$ still may have shift/reduce or reduce/reduce conflicts. $SLR(1)$ is an *improvement but not a solution.*

2.5.3 LR(1)

- **Idea**: Extend LR(0) items with a lookahead

Definition 2.5.15. (**LR(1) Items**) For the production rule $A \to \alpha\beta \in P$, for a grammar $G = (N, \Sigma, P, S)$,

$$A \to \alpha \cdot \beta, a,$$

where $a \in \mathsf{Follow}(A)$ denotes a LR(1) item.

Definition 2.5.16. (**LR(1) NFA**) Let G' be the augmented grammar of G.
Define $M(G)$ to be the NFA$^\varepsilon$ $M(G) = (\mathscr{I}(G'), N \cup \Sigma, \Delta_\varepsilon, q_0, A)$ where
$q_0 = S' \to \cdot S, \$$,
$A = \{A \to \alpha \cdot, a \in \mathscr{I}(G') : A \to \alpha \in P, a \in \mathsf{Follow}(A)\}$ and Δ_ε is defined by:

$$
\begin{aligned}
A \to \alpha \cdot X\beta, a &\xrightarrow{X} A \to \alpha X \cdot \beta, a && \text{Shift} \\
A \to \alpha \cdot B\beta, a &\xrightarrow{\varepsilon} B \to \cdot\gamma, b \in \mathsf{First}(\beta a) && \text{Examine}
\end{aligned}
$$

- Examining B is a "sub-goal" of $B\beta$, after reducing B, the next input symbol should be a prefix of β (in $\mathsf{First}(\beta)$, or a if $\gamma \implies \varepsilon$)

- Consequences:

$$
\begin{aligned}
A \to \alpha \cdot a\beta, a \in q \wedge \delta(q, a) = q' &\implies \text{shift } q' \in \mathbf{A}^G_{qa} \\
A \to \alpha \cdot, b \in q \wedge A \neq S' &\implies \text{reduce } A \to \alpha \in \mathbf{A}^G_{qb} \\
\delta(q, A) = q' &\implies \mathbf{G}^G_{qA} = q'
\end{aligned}
$$

Definition 2.5.17. ($LR(1)$) A grammar G is said to be $LR(1)$ if it's $LR(1)$ DFA $PM(G)$ is adequate.

3 Interpreters

Definition 3.0.1. (**Interpreter**) A program that executes the object language without requiring translation to a low-level representation (e.g. machine code / bytecode)

3.1 Semantics

Definition 3.1.1. ($\lambda^{\rightarrow}_{\mathbf{rec}\,+\,\mathbf{ref}\,+\,(\times/+)}$) Let Σ_{var} be a countably infinite set of variables, and Σ_{loc} be a countably infinite set of locations.

Let Σ_δ be the set of δ-functions (operators / base functions) defined by

$$\begin{aligned}\Sigma_\delta = &\{\cdot +^2 \cdot, \cdot \geq^2 \cdot\} \cup \left\{\text{fix}^2\ \cdot \cdot\right\} \\ &\cup \left\{\cdot ;^2 \cdot\right\} \cup \{\text{ref}^2 \cdot, !^1 \cdot, \cdot :=^2 \cdot\}\end{aligned}$$

and $C^n \in \Sigma_{\text{constructor}}$ is the set of constructors,

$$\begin{aligned}\Sigma_{\text{constructor}} = &\{n^0 : n \in \mathbb{Z}\} \cup \left\{\text{true}^0, \text{false}^0\right\} \cup \left\{()^0\right\} \\ &\cup \left\{(\cdot, \cdot)^2, \text{inl}^1 : \tau_1 + \tau_2, \text{inr}^1 \cdot : \tau_1 + \tau_2\right\} \\ &\cup \left\{\ell^0 : \ell \in \Sigma_{\text{loc}}\right\}\end{aligned}$$

We define the set of *primitives* as $\Sigma_{\text{primitive}} = \Sigma_{\text{constructor}} \cup \Sigma_\delta$.

The simply typed lambda calculus with recursion, references and product / sum types, denoted $\lambda^{\rightarrow}_{\text{rec}\,+\,\text{ref}\,+\,(\times/+)}$, is defined as:

$$\begin{aligned} v \ ::= \ & \lambda x : \tau.e \\ | \ & \underbrace{P^n\, v_1\ \dots\ v_n}_{\texttt{constructed values w/ arity } n} \\ | \ & \underbrace{p^n\, v_1\ \dots\ v_k}_{\texttt{partailly constructed values w/ arity } n} \qquad k < n \end{aligned}$$

$$e \ ::= \ x \in \Sigma_{\texttt{var}}$$

$$
\begin{aligned}
&\mid e_1\ e_2 \\
&\mid \lambda x:\tau.e \\
&\mid v \\
&\mid \texttt{let}\ x:\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \\
&\mid \texttt{case}\ e\ \texttt{of}\ (C_1^{m_1} x_1 : \tau_1^{C_1} \ldots x_{m_1} : \tau_n^{C_1} \to e_1 \mid \ldots \mid C_n^{m_n} x_1 : \tau_1^{C_n} \ldots x_{m_n} : \tau_m^{C_n} \to e_n)
\end{aligned}
$$

Definition 3.1.2. (**$\lambda^{\to}_{\text{rec + ref + }(\times/+)}$ Types**) The set of types $\tau \in \Sigma_\tau$ for $\lambda^{\to}_{\text{rec + ref + }(\times/+)}$ is defined by

$$
\begin{aligned}
\tau\ ::=\ &\texttt{int} \mid \texttt{bool} \mid \texttt{unit} \\
&\mid \tau_1 \to \tau_2 \mid \tau\ \texttt{ref} \\
&\mid \tau_1\ \texttt{+}\ \tau_2 \mid \tau_1 \times \tau_2
\end{aligned}
$$

- See semantics notes for *operational semantics* of $\lambda^{\to}_{\text{rec + ref + }(\times/+)}$.

3.1.1 Denotational Semantics

- **Idea**: Each syntactic construction P is associated w/ a denotation $[\![P]\!]$, a mathematical object e.g. relation, etc $\implies$ *Denotational Semantics*

Definition 3.1.3. (**Denotation of Types**) Let the universe, or *domain*, $\mathscr{U}$ be the defined by:

$$
\begin{aligned}
d\ ::=\ &n \in \mathbb{Z} \mid \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \ell \in \Sigma_{\texttt{loc}} \\
&\mid \texttt{(1, }d\texttt{)} \mid \texttt{(2, }d\texttt{)} \mid (d_1, d_2) \\
&\mid \texttt{\{((}d_1\texttt{, }s_1\texttt{), (}d_1'\texttt{, }s_1'\texttt{)), ..., ((}d_n\texttt{, }s_n\texttt{), (}d_n'\texttt{, }s_n'\texttt{))\}}
\end{aligned}
$$

The denotation function of types, denoted $[\![\cdot]\!] : \Sigma_\tau \to \mathcal{P}(\mathscr{U})$, is inductively defined by

$$
\begin{aligned}
[\![\text{int}]\!] &= \mathbb{Z} \\
[\![\text{bool}]\!] &= \{\text{true}, \text{false}\} \\
[\![\text{unit}]\!] &= \{()\} \\
[\![\tau_1 \times \tau_2]\!] &= [\![\tau_1]\!] \times [\![\tau_2]\!] \\
[\![\tau_1 + \tau_2]\!] &= [\![\tau_1]\!] \uplus [\![\tau_2]\!] \\
[\![\tau\ \text{ref}\ ]\!] &= \Sigma_\ell \\
[\![\tau_1 \to \tau_2]\!] &= \{\{((d_1, s_1), (d_1', s_1')), \ldots, ((d_n s_n), (d_n', s_n'))\} : d_i \in [\![\tau_1]\!] \implies d_i' \in [\![\tau_2]\!]\} \\
&= \mathcal{P}\left[([\![\tau_1]\!] \times \Sigma_s) \to ([\![\tau_2]\!] \times \Sigma_s)\right]
\end{aligned}
$$

Definition 3.1.4. (**Value Environment**) A value environment ρ is a finite partial function $\rho : \Sigma_{\text{var}} \rightharpoonup \Sigma_v$. ρ is well-typed in some type context Γ, denoted $\Gamma \vdash \rho$, iff

$$\operatorname{dom}\Gamma \subseteq \operatorname{dom}\rho \wedge \forall x \in \operatorname{dom}\Gamma . \rho(x) \in [\![\Gamma(x)]\!] .$$

- ρ is also known as a *closure*
- For all $\Sigma; \Gamma \vdash e : \tau$ w/ value environment $\Gamma \vdash \rho$ and store $\Sigma \vdash s$, the denotation $[\![\Sigma; \Gamma \vdash e : \tau]\!]_{\rho,s} \in [\![\tau]\!] \times \Sigma_s$

Definition 3.1.5. (**Denotation of Well-Typed Expression**) The denotation of well-typed expressions, denoted $[\![\Sigma; \Gamma \vdash e : \tau]\!]_{\rho,s}$ for $\Gamma \vdash \rho$ and $\Sigma \vdash s$ is inductively defined by:
For constructors $C^n \in \Sigma_{\text{constructor}}$:

$$\begin{aligned}
[\![\Sigma; \Gamma \vdash n : \text{int}]\!]_{\rho,s} &= (n, s) \in \mathbb{Z} \\
[\![\Sigma; \Gamma \vdash b : \text{bool}]\!]_{\rho,s} &= (b, s) \in \{\text{true}, \text{false}\} \\
[\![\Sigma; \Gamma \vdash () : \text{unit}]\!]_{\rho,s} &= ((), s) \in \{()\} \\
[\![\Sigma; \Gamma \vdash \ell : \tau \text{ ref}]\!]_{\rho,s} &= (s(\ell), s) \\
&\vdots
\end{aligned}$$

For δ-functions $\delta^n \in \Sigma_\delta$:

$$\begin{aligned}
&[\![\Sigma; \Gamma \vdash \text{fix} : [(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2] \to \tau_1 \to \tau_2]\!]_{\rho,s} \\
&= \Phi\,[\lambda f. \lambda(v_1, s_1) \in [\![\cdots]\!] \times \Sigma_s . (\lambda(v_2, s_2) \in [\![\tau_1]\!] \times \Sigma_s . \\
&\qquad \text{let } (v_3, s_3) = [v_1\ f(v_1, s_2)] \text{ in} \\
&\qquad\quad v_3\ (v_2, s_3), s_1)]
\end{aligned}$$

where Φ is a fix-point operator. (See Computation Theory).

$$\begin{aligned}
[\![\Sigma; \Gamma \vdash \text{ref} : \tau \to \tau \text{ ref }]\!]_{\rho,s} &= (\lambda(v, s') \in [\![\tau_1]\!] \times \Sigma_s . (\ell, (s', \ell \to v)), s) \\
[\![\Sigma; \Gamma \vdash ! : \tau \text{ ref} \to \tau]\!]_{\rho,s} &= (\lambda(\ell, s') \in [\![\tau \text{ ref}]\!] \times \Sigma_s . (s'(\ell), s'), s) \\
[\![\Sigma; \Gamma \vdash := : \tau \text{ ref} \to \tau \to \text{unit}]\!]_{\rho,s} &= (\lambda(\ell, s_1) \in [\![\tau \text{ ref}]\!] \times \Sigma_s . (\lambda(v, s_2) \in [\![\tau]\!] \times \Sigma_s . ((), (s_2, \ell \to v)), s_1), \\
&\vdots
\end{aligned}$$

For expressions $e \in \Sigma_e$:

$$
\begin{aligned}
[\![\Sigma;\Gamma \vdash x : \tau]\!]_{\rho,s} &= (\rho(x), s) \\
[\![\Sigma;\Gamma \vdash e_1\ e_2 : \tau_2]\!]_{\rho,s} &= ([\![\Gamma \vdash e_1 : \tau_1 \to \tau_2]\!]_{\rho,s})([\![\Gamma \vdash e_2 : \tau_1]\!]_{\rho,s}) \\
[\![\Sigma;\Gamma \vdash \text{let } x : \tau_1 = e_1 \text{ in } e_2 : \tau_2]\!]_{\rho,s} &= (\lambda(v, s') \in [\![\tau_1]\!] \times \Sigma_s.\, [\![\Sigma;\Gamma \vdash e_2 : \tau_2]\!]_{(\rho, x \to v), s'})\, [\![\Sigma;\Gamma \vdash e_1 : \tau_1]\!]_{\rho,s} \\
[\![\Sigma;\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2]\!]_{\rho,s} &= (\lambda(v, s_1) \in [\![\tau_1]\!] \times \Sigma_s.\, [\![\Sigma;\Gamma \vdash e : \tau_2]\!]_{(\rho, x \to v), s_1}, s) \\
[\![\Sigma;\Gamma \vdash \text{case } e \ldots : \tau']\!]_{\rho,s} &= [\![\Sigma;\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_{m_i} \vdash e_i : \tau']\!]_{(\rho, x_i \to v_i), s'} \\
&\quad \text{where } (C_i^{m_i}\ v_1\ \ldots\ v_{m_i}, s') = [\![\Sigma;\Gamma \vdash e : \tau]\!]_{\rho,s}
\end{aligned}
$$

3.1.2 Definitional Interpreter I_0

- **Idea**: Denotation semantics yield a *meta-circular* or *definitional* interpreter.

Definition 3.1.6. (**Meta-Circular Interpreter**) An interpreter for an object language $\mathcal{L}$ defined in some meta-language $\mathcal{L}'$ where the features of $\mathcal{L}$ are defined using the corresponding feature of the meta-language.

- e.g. function application in $\lambda^{\to}$ implemented using application in `OCaml`

3.1.2.1 Syntax

```
module Identifier = struct
  type t = string
  let show x = x
end

module Constructor = struct
  type t =
    | CInt of int | CBool of bool | CUnit
    | CInl | CInr
    | CPair

  let show c = ...
end

module Delta = struct
```

```
    type t =
        | DFix
        | DAdd | DGeq | DSeq
        | DRef | DDeref | DAssign

    let show δ = ...
end

module Primitive = struct
    type t =
        | Delta of Delta.t
        | Constructor of Constructor.t

    let show p = ...
end

module Expr = struct
    type t =
        | EPrimitive of Primitive.t
        | EVar of Identifier.t
        | EApp of t * t
        | ELambda of Identifier.t * Type.t * t
        | ELet of Identifier.t * Type.t * t * t
        | ECase of t * branch list
    and branch = Constructor.t * (Identifier.t * Type.t) list * t

    let show e = ...
end
```

3.1.2.2 Interpreter

- Denotational representations:

```
module Location = struct
    type t = int
    let show = Int.string_of

    (* used by the ref δ-function *)
    let new =
```

```
        let current = ref 0
        in λ () → current := !current + 1; !current
end

module Value = struct
    type t =
        | VInt of int | VBool of bool | VUnit
        | VRef of Location.t
        | VPair of t * t
        | VInl of t | VInr of t
        | VFunction of (t * Store.t) → (t * Store.t)

    let apply v₁ (v₂, s) = match v₁ with
        | VFunction f -> f (v₂, s)
        | _ -> raise RuntimeError "Expecting Function"

    let show v = ···
end

module Environment = struct
    type t = Identifier.t → Value.t

    let empty = λ x →
        raise RuntimeError ("undefined variable " ^ Identifier.show x)
    let extend ρ x v =
        λ y → if y = x then v else ρ y
end

module Store = struct
    type t = Location.t → Value.t

    let empty = λ ℓ →
        raise RuntimeError ("undefined location " ^ Location.show ℓ)
    let extend s ℓ v =
        λ ℓ′ → if ℓ′ = ℓ then v else s ℓ
end
```

- Interpreters for `Constructors` and `Delta`-functions:

```
module Constructor = struct
    let interpret c = match c with
        | CInt n -> ...
        | CInl -> VFunction (λ (v1, s1) -> (VInl v1, s1))
        | CInr -> VFunction (λ (v1, s1) -> (VInr v2, s2))
        | CPair -> VFunction
            (λ (v1, s1) -> (VFunction
                (λ (v2,s2) -> (VPair (v1,v2), s2)), s1))
end

module Delta = struct
    let interpret δ = match δ with
        | ...
        | DFix ->
            let rec fix v1 s1 = VFunction
                (λ (v2, s2) ->
                    (Value.apply (Value.apply v1 (fix v1 s2)) v2, s2)
                , s1)
            in VFunction fix
end

module Primitive = struct
    let interpret p = match p.body with
        | `Delta δ -> Delta.interpret δ
        | `Constructor c -> Constructor.interpret c
end
```

- Interpreter I_0:

```
let rec interpret e ρ s = match e with
    | EPrimitive p -> (Primitive.interpret p, s)
    | EVar x -> (ρ x, s)
    | EApp (e1, e2) ->
        let (v1, s1) = interpret e1 ρ s
        and (v2, s2) = interpret e2 ρ s1 in
            Value.apply v1 (v2, s2)
    | ELambda (x, _, e) ->
```

```
    (VFunction (λ (v, s) ->
        let ρ' = Environment.extend ρ x v
        in interpret e ρ' s), s)
| ELet (x, e₁, _, e₂) ->
    let (v₁, s₁) = interpret e₁ ρ s
    and ρ' = Environment.extend ρ x v₁ in
        interpret e₂ ρ' s₁
| ECase (e, bs) ->
    let (v₁,s₁) = interpret e₁ ρ s in
    and (match matches v₁ bs with
        | Some (e, bindings) ->
            let ρ' = List.fold_left
                (λ ρ (x, v) -> Environment.extend ρ x v) ρ bindings
            in interpret e ρ' s₁
        | None -> raise RuntimeError "Insufficient Cases")

let matches v = List.find_map (λ (c, xs, e) -> match c, xs, v with
    | ...
    | CPair, [x₁; x₂], VPair (v₁, v₂) -> Some (e, [(x₁,v₁); (x₂,v₂)])
    | _, _ -> None)
```

- **Problem**:
  - Semantic dependence on the meta-language $\implies$ evaluation strategies / features of meta-language must match the object language.
  - Adding features not defined in the meta-language is difficult / impossible.
- **Solution**: CPS to explicitly define evaluation strategy / control flow + defunctionalization for low level representations (not dependent on higher order functions)

3.2 Transformations

3.2.1 Continuation Passing Style

- **Problem**: Definitional interpreters suffer from insufficient control flow for *non-local control flow features* such as errors, gotos, etc.

- For an arbitrary expression in context E, $E[e]$, we transform E into a continuation, a λ-function of the form: $(\lambda x.E[x])e$.
- $\lambda x.E[x]$ is a *continuation*, which specifies exactly what a expression to evaluate once e has been evaluated $\implies$ CPS λ-calculus

Definition 3.2.1. (**CPS λ-Calculus**) The syntax of CPS λ-calculus is defined by:

$$\begin{aligned} v &::= x \mid \lambda x.e \\ e &::= v_1\ v_2 \end{aligned}$$

with denotational universe $\mathscr{U}$ defined by

$$d ::= * \mid \{(d_1, d_1'), \ldots, (d_n, d_n')\}$$

and denotational semantics $\mathscr{V}\,[\![\cdot]\!]_\rho : \Sigma_v^{CPS} \to \mathscr{U}, \mathscr{E}\,[\![\cdot]\!]_\rho : \Sigma_e^{CPS} \to \mathscr{U}$ with a well-defined value environment $e \vdash \rho$ (or $v \vdash \rho$) $\iff$ $\operatorname{dom}\rho \subseteq fv(e)$ (or $fv(v)$):

$$\begin{aligned} \mathscr{V}\,[\![x]\!]_\rho\,(k) &= \rho(x) \\ \mathscr{V}\,[\![\lambda x.e]\!]_\rho &= \lambda v \in \mathscr{U}.\mathscr{E}\,[\![e]\!]_{(\rho, x \to v)} \\ \mathscr{E}\,[\![v_1\ v_2]\!]_\rho &= \mathscr{V}\,[\![v_1]\!]_\rho\ \mathscr{V}\,[\![v_2]\!]_\rho \end{aligned}$$

The operational semantic transition relation $e \longrightarrow e'$ defined by

$$(\lambda x.e)\ v \longrightarrow \{v/x\}\,e$$

- Simplicity of $\longrightarrow$ and mutual recursion of $\mathscr{V}$ and $\mathscr{E}$ results in a tail recursive interpreter $\implies$ CPS is "lower-level" than λ-calculus.

Definition 3.2.2. (**CPS Conversion**) The CPS conversion function, denoted $[\![\cdot]\!]\,(\cdot) : \Sigma_e^\lambda \to (\Sigma_v^{CPS} \to \Sigma_e^{CPS})$ is defined by

$$\begin{aligned} [\![x]\!]\,(k) &= k\ x \\ [\![\lambda x.e]\!]\,(k) &= k\ (\lambda k' x.\,[\![e]\!]\,(k')) \\ [\![e_0\ e_1]\!]\,(k) &= [\![e_0]\!]\,(\lambda f.\,[\![e_1]\!]\,(\lambda x.(f\ k)\ v)) \end{aligned}$$

where k is an explicit *continuation*.

Theorem 3.2.1. (**Equivalence**)

$$\forall e \in \Sigma_e^\lambda, v \in \Sigma_v^\lambda.$$
$$e \longrightarrow^* v \iff \forall k \notin fv(e). \llbracket e \rrbracket\, k \longrightarrow^* \llbracket v \rrbracket\, k$$

- **Consequence**: Every program may be written in a CPS form.
- **Example**:

```
let (v1, s1) = interpret e1 ρ s
and (v2, s2) = interpret e2 ρ s1 in
    (match v1 with
    | VFunction f -> f (v2, s2)
    | _ -> raise RuntimeError "Expecting function")
```

translates to

```
interpret e1 ρ s (λ (v1, s1) ->
    interpret e2 ρ s1 (λ (v2, s2) ->
        match v1 with
        | VFunction f -> f (v2, s2, k)
        | _ -> raise RuntimeError "Expecting Function"))
```

Note the passing of the continuation to the function f, since f may call `interpret` (See `ELambda` case).

3.2.1.1 Interpreter I_1

- CPS yields Interpreter I_1:

```
module Continuation = struct
    type t = (Value.t * Store.t) -> (Value.t * Store.t)
end

module Value = struct
    type t =
        | ...
        | VFunction of (t * Store.t * Continuation.t) -> (t * Store.t)

    let apply v1 (v2, s, cnt) = match v1 with
        | VFunction f -> f (v2, s, cnt)
```

```
        | _ -> raise RuntimeError "Expecting Function")
end

module Delta = struct
    let interpret δ = match δ with
        | ...
        | DFix ->
          let rec fix (v1, s1, cnt1) = cnt1
            (VFunction
              (fun (v2, s2, cnt2) ->
                fix (v1, s2,
                  (fun (v3, s3) ->
                    Value.apply v1 (v3, s3,
                      (fun (v4, s4) ->
                        Value.apply v4 (v2, s4, cnt2)))))),
            s1)
          in VFunction fix
end

let interpret = interpret' (λ x -> x)
let rec interpret' e ρ s cnt = match e with
    | EPrimitive p -> cnt (v, s)
    | EVar x -> cnt (ρ x, s)
    | EApp (e1, e2) ->
        interpret' e1 ρ s (λ (v1, s1) ->
            interpret' e2 ρ s1 (λ (v2, s2) ->
                Value.apply v1 (v2, s2, cnt)))
    | ELambda (x, _, e) ->
        cnt (VFunction (λ (v, s, cnt) ->
                let ρ' = Environment.extend ρ x v
                in interpret' e ρ' s cnt)
            , s)
    | ELet (x, e1, _, e2) ->
        interpret' e1 ρ s (λ (v1, s1) ->
            let ρ' = Environment.extend ρ x v1 in
                interpret' e2 ρ' s1 cnt)
    | ECase (e, bs) ->
        interpret' e1 ρ s (λ (v1, s1) ->
```

```
match matches v₁ bs with
    | Some (e, bindings) ->
        let ρ' = List.fold_left
            (λ ρ (x, v) -> Environment.extend ρ x v) ρ bindings
        in interpret' e ρ' s₁ cnt
    | None -> raise RuntimeError "Insufficient Cases")
```

- **Observation**: CPS translation yields an entirely *higher-order tail recursive* function $\implies$ Interpreter may be implemented using an iterative loop (See section ??)

3.2.2 Defunctionalization

- **Problem**: Interpreters I_0 and I_1 rely on higher-order functions to represent the denotation of function values (`VFunction`).
- **Solution**: Defunctionalization

Definition 3.2.3. (**Defunctionalized CPS λ-calculus**) The Defunctionalization of the CPS λ-calculus has the denotational universe $\mathscr{U}$:

```
d ::= (ρ, x, e)
```

and denotational semantics

$$\begin{aligned}
\mathscr{V}\,[\![x]\!]_\rho &= \rho(x) \\
\mathscr{V}\,[\![\lambda x.e]\!]_\rho &= (\rho, x, e) \\
\mathscr{E}\,[\![v_1\ v_2]\!]_\rho &= \mathscr{E}\,[\![e]\!]_{(\rho', x \to \mathscr{V}[\![v_2]\!])} \text{ where } (\rho', x, e) = \mathscr{V}\,[\![v_1]\!]
\end{aligned}$$

Theorem 3.2.2. (**Equivalence**)

$$\begin{aligned}
&\forall e \in \Sigma_e. \forall \rho \in \Sigma_\rho \\
&\qquad e \vdash \rho \implies \mathscr{E}\,[\![e]\!]_\rho^{CPS} \simeq \mathscr{E}\,[\![e]\!]_\rho^{DFN}
\end{aligned}$$

where $\simeq: \mathscr{U}^{CPS} \longleftrightarrow \mathscr{U}^{DFN}$ is a universe equivalence relation.

- Defunctionalization (DFN) is a transformation that eliminates higher order functions:

  $$\begin{aligned}
  [\![\Gamma_i \vdash \lambda x : \tau_1.e_i : \tau_1 \to \tau_2]\!] &= \mathrm{F}_i^{\tau_1,\tau_2}\ (x_1, \ldots, x_n) \text{ where } x_i \in fv(e_i) \setminus \{x\} \\
  [\![\Gamma \vdash e_1\ e_2 : \tau_2]\!] &= \mathrm{apply}_{\tau_1,\tau_2}\ [\![\Gamma \vdash e_1 : \tau_1 \to \tau_2]\!]\ \ [\![\Gamma \vdash e_2 : \tau_1]\!]
  \end{aligned}$$

  where

```
let apply_{τ1,τ2} f y = match f with
    | F_i^{τ1,τ2} -> ⟦Γ_i, x : τ1 ⊢ {y/x} e_i : τ2⟧
    | ...

type function_{τ1,τ2} =
    | F_i^{τ1,τ2} of ∏_{x∈fv(e_i)\{x}} τ_x [where (x, τ_x) ∈ Γ_i]
    | ...
```

- **DFN Conversion**:
  1. For each of the higher-order functions $f = \lambda x : \tau_1.e_i$ to be replaced:
     (a) Compute the free variables (or *closure*) of f: $fv(f) = \{x_1, \ldots, x_n\}$ w/ types $\tau_1, \ldots, \tau_n$
     (b) Define a constructor w/ type $\texttt{F}_i$ `of` τ_1 `* ... *` τ_n
  2. Define an `apply` function:

     ```
     let apply f y = match f with
         | F_i (x_i, ..., x_{n_i}) -> ⟦{y/x} e_i⟧
         | ...
     ```

     where $\texttt{F}_i$ represents the higher-order function $\lambda x : \tau_1.e_i$ and $fv(e_i) \setminus \{x\} = \{x_1, \ldots, x_{n_i}\}$
  3. Replace all applications w/ `apply`.

- **Example**: Defunctionalizing `Environment.`t:
  1. We have the following analysis of `Environment.`t's higher-order functions:

Higher-Order Function	Closure	Constructor
λ x `-> raise RuntimeError ...`	$\emptyset$	`Empty`
λ y `-> if` $y = x$ `then` v `else` ρ y	$\{\rho, x, v\}$	`Extend (`x`,` v`,` ρ`)`

Yields the ADT type definition:

```
type t =
    | Empty
    | Update of Identifier.t * Value.t * t
```

2. The `apply` (renamed to *lookup* for semantic meaning) function is implemented as:

```
let rec lookup ρ y = match ρ with
    | Empty -> raise RuntimeError ("undefined variable" ... )
    | Update (x, v, ρ') -> if y = x then v else lookup ρ' y
```

- Isomorphisms between ADT's may result in further transformations. e.g. `Environment.t` is isomorphic to `(Identifier.t * Value.t) list`

3.2.2.1 Interpreter I_2

- **Idea**: Defunctionalize `Store`, `Environment`, `Value` and *continuations* in $I_1 \implies I_2$.
- Defunctionalized Continuation type:

```
type t =
    | CId
    | CApp1 of Expr.t * Environment.t * t
    | CApp2 of Value.t * t
    | CLet1 of Identifier.t * Environment.t * Expr.t * t
    | CCase1 of Expr.branch list * Environment.t * t
```

is isomorphic to:

```
type t = action list
and action =
    | CApp1 of Expr.t * Environment.t
    | CApp2 of Value.t
    | CLet1 of Identifier.t * Environment.t * Expr.t
    | CCase1 of Expr.branch list * Environment.t
    | CFix1 of Value.t * Value.t
    | CFix2 of Value.t
```

- **Note**: This explicitly makes continuations a *stack representation*.

```
module Function = struct
    type t =
        | FClosure of Identifier.t * Expr.t * Environment.t
```

```
        | FFix1
        | FFix2 of Value.t
        | ...

    let rec apply f (v, s, cnt) = match f with
        | FClosure (x, e, ρ) ->
          let ρ' = Environment.extend ρ x v in
              interpret' e ρ s cnt
        | FFix1 -> Continuation.apply cnt (VFunction (FFix2 v), s)
        | FFix2 v1 -> apply FFix1 (v1, s, CFix1 (v1, v) :: cnt)
        | ...
end

module Value = struct
    type t =
        | ...
        | VFunction of Function.t

    let apply v1 (v2, s, cnt) = match v1 with
        | VFunction f -> Function.apply f (v2, s, cnt)
        | _ -> raise RuntimeError "Expecting Function")
end

module Continuation = struct
    type t = cnt_action list
    and action =
        | CApp1 of Expr.t * Environment.t
        | CApp2 of Value.t
        | CLet1 of Identifier.t * Environment.t * Expr.t
        | CCase1 of Expr.branch list * Environment.t
        | CFix1 of Value.t * Value.t
        | CFix2 of Value.t

    let rec apply cnt (v, s) = match cnt with
        | [] -> (v, s)
        | CApp1 (e1, ρ) :: cnt -> interpret' e1 ρ s (App2 (v, cnt))
        | CApp2 v1 :: cnt -> Value.apply v1 (v2, s, cnt)
        | CLet1 (x, ρ, e2) :: cnt ->
```

```
          let ρ' = Environment.extend ρ x v in
            interpret' e₂ ρ' s cnt
        | CCase1 (bs, ρ) :: cnt ->
          (match matches v bs with
          | Some (e, bindings) ->
            let ρ' = List.fold_left
                (λ ρ (x, v) -> Environment.extend ρ x v) ρ bindings
            in interpret; e ρ' s cnt
          | None -> raise RuntimeError "Insufficient Cases")
        | CFix1 (v₁, v₂) :: cnt -> Value.apply v₁ (v, s, CFix2 v₂ :: cnt)
        | CFix2 v₂ :: cnt -> Value.apply v (v₂, s, cnt)
end

let rec interpret' e ρ s cnt = match e with
    | EPrimitive p -> Continuation.apply cnt (Primitive.interpret p, s)
    | EVar x -> Continuation.apply cnt (Environment.lookup ρ x, s)
    | EApp (e₁, e₂) ->
      interpret' e₁ ρ s (CApp1 (e₂, ρ) :: cnt)
    | ELambda (x, _, e) ->
      Continuation.apply cnt (VFunction (FClosure (x, e, ρ)), s)
    | ELet (x, e₁, _, e₂) ->
      interpret' e₁ ρ s (CLet1 (x, ρ, e₂) :: cnt)
    | ECase (e, bs) ->
      interpret' e ρ s (CCase1 (bs, ρ) :: cnt)
```

- **Problem**: `interpret'`, `Continuation.apply`, `Function.apply` are all mutually recursive

3.2.3 Tail and Mutual Recursion Elimination

- **Problem**: Recursion is a high-level functional construct of the interpreters I_0, I_1, I_2. Desire to use iterative structures for efficiency (removes stack frames) and lower level applications (VMs).

Definition 3.2.4. (**Mutually Recursive**) Mutual recursion is where two functions (or datatypes) are defined in terms of each other. In OCaml, they have the form:

let rec f_1 x_1 ... x_{m_1} = e_1
and ...
and f_n x_1 ... x_{m_n} = e_n

- **Example**:

  let rec is_even n = if $n = 0$ then true else is_odd $(n-1)$
  and is_odd n = if $n = 0$ then false else is_even $(n-1)$

  or `interpret'`, `Continuation.apply`, `Function.apply` (see section ??).

- **Elimination Transformation**:

  1. Define a type `action` containing the arguments of each function

     type action =
     | F_1 of τ_1^1 * ... * $\tau_{m_1}^1$
     | F_2 of τ_1^2 * ... * $\tau_{m_2}^2$
     | ...
     | F_n of τ_1^n * ... * $\tau_{m_n}^n$

  2. Uncurry all applications of $f_1, \ldots, f_n$ in $e_1, \ldots, e_n$.
  3. Define the function:

     let rec f action = match action with
     | F_1 $(x_1, \ldots, x_{m_1})$ -> $[\![e_1]\!]$
     | ...
     | F_n $(x_1, \ldots, x_{m_n})$ -> $[\![e_n]\!]$

     where

     $$[\![f_j\ (e_1, \ldots, e_{m_j})]\!] = f\ (\mathrm{F}_j\ (e_1, \ldots, e_{m_j})) \qquad \text{Replace}$$
     $$[\![e]\!] = e \qquad \text{Other}$$

     $[\![e_i]\!]$ is e_i w/ all occurrences of $f_j(e'_1, \ldots, e'_{m_j})$ replaced w/ f (F_j $(e'_1, \ldots, e'_{m_j})$).

- If e_i are *simple mutual tail recursive*, then $[\![e_i]\!]$ are *simple tail recursive*. Thus we can factor the simple tail recursion into a `driver` function and f yields a *state transition function* `step`.

Definition 3.2.5. (**Tail Recursive**) A function f is said to be tail recursive if the last operation of f is a recursive call to f.

f is said to be *simple tail recursive* if $f \notin fv(e_i)$ where e_i is an expression for an argument applied to f.

- **Observation**:
  - CPS transforms all functions into a tail recursive function w/ a continuation
  - Defunctionalization on the continuation types defines an *explicit* stack based datastructure w/ a mutually recursive `apply` function
  - Mutual recursion elimination defines a state machine (on `action`s) which converts a set of mutually recursive functions (using DFN CPS) into a non-recursive `step` function w/ a *simple tail recursive* `driver` function. (See section ??)
- **Elimination Transformation** (*for simple tail recursion*):
  - Write the simple tail recursive function f in the form:

```
let rec f x_1 ... x_n = match e_1, ..., e_m with
    | p^i_1, ..., p^i_m -> b_i
    | ...
    | p^j_1, ..., p^j_m -> r_j
```

    where e_k, b_i, r_j are expressions, satisfying $f \notin fv(e_k)$, $f \notin fv(b_i)$ a *base case* and $f \in fv(r_j)$ and r_j is in *simple tail recursive* form.
  - Define a break exception for loops: `exception Break`. The iterative form of f is given by:

```
exception Break
let f y_1 ... y_n =
    let x_1 = ref y_1
    and ...
    and x_n = ref y_n
    and r = ref None in
        try (while (true) do (
                match [[e_1]], ..., [[e_m]] with
                    | p^i_1, ..., p^i_m -> r := Some ([[b_i]]); raise Break
```

```
                    | ...
                    | p_1^j, ..., p_m^j -> [[r_j]])
          with Break -> ();
          Option.get !r
```

where

$$
\begin{aligned}
[\![f\ e_1\ \dots e_n]\!] &= x_1 := [\![e_1]\!]\,; \dots ; x_n := [\![e_n]\!] && \text{Set Arguments} \\
[\![x_i]\!] &= !x_i && \text{Dereference arguments} \\
[\![e]\!] &= e && \text{Other}
\end{aligned}
$$

3.2.3.1 Interpreter I_3

- Eliminating mutual tail recursion from I_2 yields I_3 w/ the `step` function:

```
type state_action =
    | FApply of Function.t * (Value.t * Store.t)
    | CApply of (Value.t * Store.t)
    | Interpret of Expr.t * Environment.t * Store.t
and state = Continuation.t * state_action

let step = function
    | (cnt, Interpret (EPrimitive p, ρ, s))
       -> (cnt, CApply (Primitive.interpret p, s))
    | (cnt, Interpret (EVar x, ρ, s))
       -> (cnt, CApply (Environment.lookup ρ x, s))
    | (cnt, Interpret (EApp (e1, e2), ρ, s))
       -> (CApp1 (e2, ρ) :: cnt, Interpret (e1, ρ, s))
    | (cnt, Interpret (ELambda (x, _, e), ρ, s))
       -> (cnt, CApply (VFunction (FClosure (x, e, ρ)), s))
    | (cnt, Interpret (ELet (x, e1, _, e2), ρ, s))
       -> (CLet1 (x, ρ, e2) :: cnt, Interpret (e1, ρ, s))
    | (cnt, Interpret (ECase (e, bs), ρ, s))
       -> (CCase1 (bs, ρ) :: cnt, Interpret (e, ρ, s))

    | (cnt, FApply (FClosure (x, e, ρ), v, s)) ->
       let ρ' = Environment.extend ρ x v in
           (cnt, Interpret (e, ρ', s))
```

```
    | (cnt, FApply (FFix1, v, s))
        -> (cnt, CApply (VFunction (FFix2 v), s))
    | (cnt, FApply (FFix2 v1, v, s))
        -> (CFix1 (v1, v) :: cnt, FApply (FFix1, v1, s))
    | ...

    | ([], CApply (v, s)) -> ([], CApply (v, s))
    | ...

let rec driver = function
    | ([], CApply (v, s)) -> (v, s)
    | s -> driver (step s)

let interpret e ρ s = driver ([], Interpret (e, ρ, s))
```

- `step` is a non-recursive *state transition function* and `driver` is a *simple tail recursive* function, with an explicit *runtime-state cnt*. A *high-level stack machine.*
- **Observation**: We can split the `state_action` type into a *stack* of *values* and a stack of environments, factoring *store* into the state:
  - `CApply` (v, s) corresponds to popping / pushing a value v onto the stack vs
  - `Interpret` $(e,\ \rho,\ s)$ corresponds to popping / pushing environment ρ onto ρs

```
module Continuation = struct
    type t =
        // | CApp1 of Expr.t factored out by a sequence of transitions from EA
        | CApp2
        | CLet1 of Identifier.t * Expr.t
        | CCase1 of Expr.branch list
        | CFix1
        | CFix2

    ...
end
```

```
type action =
    | Interpret of Expr.t
    | CApply of Continuation.t
    | FApply
    | PopEnvironment // due to environment stack semantics
and state = action list * Value.t list * Environment.t list * Store.t

let step = function
    | (Interpret (EPrimitive p) :: as, vs, ρs, s) ->
        (as, (Primitive.interpret p) :: vs, ρs, s)
    | (Interpret (EVar x) :: as, vs, ρ :: ρs, s)
        -> (as, (Environment.lookup ρ x) :: vs, ρ :: ρs, s)
    | (Interpret (EApp (e1, e2)) :: as, vs, ρs, s)
        -> (Interpret e1 :: Interpret e2 :: CApply CApp2 :: as, vs, ρs, s)
    | (Interpret (ELambda (x, _, e)) :: as, vs, ρ :: ρs, s)
        -> (as, VFunction (FClosure (x, e, ρ)) :: vs, ρ :: ρs, s)
    | (Interpret (ELet (x, _, e1, e2)) :: as, vs, ρs, s)
        -> (Interpret e1 :: CApply (CLet1 (x, e2)) :: as, vs, ρs, s)
    | (Interpret (ECase (e, bs)) :: as, vs, ρs, s)
        -> (Interpret e :: CApply (CCase1 bs) :: as, vs, ρs, s)

    | (FApply :: as, v :: VFunction (FClosure (x, e, ρ)) :: vs, ρs, s) ->
        let ρ' = Environment.extend ρ x v in
          (Interpret e :: PopEnvironment :: as, vs, ρ' :: ρs, s)

    | ...

    | (PopEnvironment :: as, vs, ρ :: ρs, s) -> (as, vs, ρs, s)
```

- **Observation**: `step` consists of an `Interpret` phase and then an `Apply` phase interleaved. `Interpret` translates the `Expr.`t into a *stack of actions* (*instructions*)
- **Idea**: Pre-compute the `Interpret` phase using a function `compile` $\Longrightarrow$ *compiling*

4 Compilers

Definition 4.0.1. (**Compiler**) A program that translates (compiles) the object language into a low-level representation (e.g. machine code / byte-code)

4.1 Compiler C_0

- **Idea**: Precomputing the `Interpret` phase of the `step` function in I_3 yields compiler C_1.
- Defunctionalized *continuations / actions* form the *instruction set* of the stack machine. Additional instructions required for *environments*.

```
module Instruction = struct
   type t =
      | IPush of Value.t // Derived from Interpret (EPrimitive p)

      | IExtend of Identifier.t // Derived from CLet1 continuation
      | ILookup of Identifier.t // Derived from Interpret (EVar x)

      | IMakeClosure of (t list) // Derived from Interpret (ELambda (x, _, e))
      | IPopEnvironment

      | ICase of (Constructor.t * t list) list
      | IFix1
      | IFix2
      | IApp // Alias for CApp2

   let show i = ...
end
```

```
let rec compile = function
    | EPrimitive p -> [IPush (Primitive.interpret p)]
    | EVar x -> [ILookup x]
    | EApp (e1, e2) ->
        (compile e1)
        @ (compile e2)
        @ [IApp]
    | ELambda (x, _, e) ->
        [IMakeClosure (
          (IExtend x)
          :: (compile e)
          @ [IPopEnvironment])]
    | ELet (x, _, e1, e2) ->
        (compile e1)
        @ [IExtend x]
        @ (compile e2)
        @ [IPopEnvironment]
    | ECase (e, bs) ->
        (compile e)
        @ [ICase (List.map (fun (c, bs, e) ->
            let exs = List.map (fun (x, _) -> IExtend x) bs in
            let is = exs
              @ (compile e)
              @ [IPopEnvironment] in
                (c, is)) bs)]
```

- The highlevel *stack machine* for C_0:

```
type state = Instruction.t list * Value.t list * Environment.t list * Store.t

let step = function
    | (IPush v :: is, vs, ρs, s)
        -> (is, v :: vs, ρs, s)
    | (IApp :: is, v :: VFunction f :: vs, ρs, s)
        -> Function.apply f v (is, vs, ρs, s)
    | (IExtend x :: is, v :: vs, ρ :: ρs, s)
        -> (is, vs, Environment.extend ρ x v :: ρs, s)
```

```
| (ILookup x :: is, vs, ρ :: ρ, s)
    -> (is, Environment.lookup ρ x :: vs, ρ :: ρ, s)
| (IMakeClosure is₁ :: is₂, vs, ρ :: ρs, s)
    -> (is₂, VFunction (FClosure (is₁, ρ)) :: vs, ρ :: ρs, s)
| (IPopEnvironment :: is, vs, _ :: ρs, s) -> (is, vs, ρs, s)
| ...
| (ICase ciss :: is, v :: vs, ρ :: ρs, s) ->
    (match matches v ciss with
    | Some (is₁, vs₁) -> (is₁ @ is, vs₁ @ vs, ρ :: ρ :: ρs, s)
    | None -> raise RuntimeError "Unmatched case")
```

- **Problems**:
  - Inefficient environments (duplication for *immutability*) via the environment stack
  - Instruction set is *highlevel* and "tree-like" (containing nested instructions) e.g. `IMakeClosure` or `ICase`

4.2 Compiler C_1

4.2.1 Linearizing Instructions

- **Problem**: Instructions are "tree-like", containing `Instruction.`t *lists* (e.g. `IMakeClosure`).
- **Solution**: Linearize instructions using *addresses*
- At *runtime*: Instructions are stored in an *addressable array* w/ a *code pointer* which stores the index of the *next instruction*

```
module Label = struct
  type t = int

  let fresh = ...
  ...
end
```

```
module Instruction = struct
    type t =
        | ...
        | IJump of Label.t
        | ILabel of Label.t
        | IReturn

    ...
end
```

- Requires additional instructions:
    - `IJump` l: Jump to address associated with label l
    - `IReturn`: Pop return address a and set code pointer to a
    - `ILabel` l: Associate label l w/ current address (at runtime, used for linking / loading)
- **Problem**: Linearizing instructions requires splitting instructions list is into instructions for (function) *definitions* (ds) and the main *program instructions* (is).
- The pair (ds, is) is defined as a *bytecode blob* (or *object*).

```
module Bytecode = struct
    type t = Instruction.t list * Instruction.t list

    // Chain bytecode
    let ( @> ) (ds1, is1) (ds2, is2) = (ds1 @ ds2, is1 @ is2)
    let pure x = ([], x)
end

let rec compile = function
    | EPrimitive p -> pure [IPush (Primitive.interpret p)]
    | EVar x -> pure [ILookup x]
    | EApp (e1, e2) ->
        (compile e1)
        @> (compile e2)
        @> (pure [IApp])
```

```
  | ELambda (x, _, e) ->
      let l = Label.fresh ()
      and (ds, is) = compile e in
      let d =
          (pure [ILabel l; IExtend x])
          @> is
          @> (pure [IPopEnvironment; IReturn]) in
              (d @ ds, [IMakeClosure l])
  | ELet (x, _, e1, e2) ->
      (compile e1)
      @> (pure [IExtend x])
      @> (compile e2)
      @> (pure [IPopEnvironment])
  | ECase (e, bs) ->
      let lbs = List.map (λ b -> (Label.fresh (), b)) bs
      and l = Label.fresh () in
          (compile e)
          @> (pure [ICase (List.map (λ (l, (c, _, _)) -> (c, l)))])
          @> (List.flat_map (λ (lc, (c, bs, e)) ->
                  (pure [ILabel lc])
                  @> (pure (List.map (fun (x, _) -> IExtend x) bs))
                  @> (compile e)
                  @> (pure [IPopEnvironment; IJump l]))
              lbs)
          @> (pure [ILabel l])
```

4.2.2 Loading

- **Problem**: Labels must be resolved to addresses before execution *at runtime*
- **Solution**: *Loading*. Requires additional instruction `IHalt`.

```
module Label_map = struct
    ...
    let of_instructions = List.fold_right (λ i (cp, las) -> match i with
        | ILabel l -> (cp + 1, Label.assoc_add (l, cp) las)
        | _ -> (cp + 1, las))
```

```
end

module Label = struct
    ...
    let resolve las = function
        | IJump l -> IJump (List.assoc l las)
        | ICase cls -> ICase (List.assoc_map (λ l -> List.assoc l las))
        | IMakeClosure l -> IMakeClosure (List.assoc l las)
end

let load (ds, is) =
    let is' = is @ IHalt :: ds in
    let las = Label_map.of_instructions is' in
        Array.of_list (List.map (Label.resolve las) is')
```

4.2.3 Control Flow

- Control flow requires additional runtime state:
  - *Code pointer*: An address to the next instruction
  - *Return address stack*: A stack of return addresses that are popped and pushed by `IReturn` and `IApp`.
- C_1 stack machine:

```
type state = Instruction.t array * int
    * Value.t list * Environment.t list * int list
    * Store.t

module Function = struct

    let apply f v (is, cp, vs, ρs, ras, s) = match f with
        | FClosure (cp', ρ) -> (is, cp', v :: vs, ρ :: ρs, cp :: ras, s)
        | ...

end
```

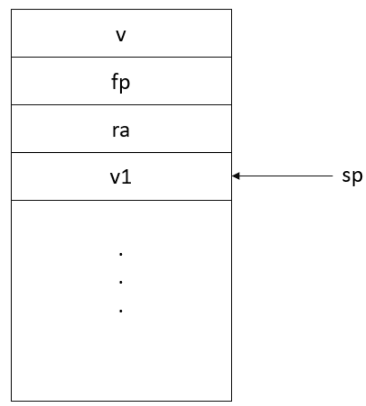
```
let step (is, cp, vs, ρs, ras, s) = match (is.(cp), vs, ρs, ras) with
    | (IPush v, vs, ρs, ras)
        -> (is, cp + 1, v :: vs, ρs, ras, s)
    | (IApp, v :: VFunction f :: vs, ρs, ras)
        -> Function.apply f v (is, cp + 1, vs, ρs, ras, s)
    | (IExtend x, v :: vs, ρ :: ρs, ras)
        -> (is, cp + 1, vs, Environment.extend ρ x v :: ρs, ras, s)
    | (ILookup x :: is, vs, ρ :: ρs, ras)
        -> (is, cp + 1, Environment.lookup ρ x :: vs, ρ :: ρs, ras, s)
    | (IMakeClosure l, vs, ρ :: ρs, ras)
        -> (is, VFunction (FClosure (l, ρ)) :: vs, ρ :: ρs, ras, s)
    | (IPopEnvironment, vs, _ :: ρs, ras) -> (is, cp + 1, vs, ρs, ras, s)
    | ...
    | (ICase cls, v :: vs, ρ :: ρs, ras) ->
        (match matches v cls with
            | Some (l_c, vs_c) -> (is, l_c, vs_c @ vs, ρ :: ρ :: ρs, ras, s)
            | None -> raise RuntimeError "Unmatched case")
    | (IReturn, vs, ρs, ra :: ras) -> (is, ra, vs, ρs, ras, s)
    | (ILabel l, vs, ρs, ras) -> (is, cp + 1, vs, ρs, ras, s)
    | (IJump l, vs, ρs, ras) -> (is, l, vs, ρs, ras, s)
    | (IHalt, vs, ρs, ras) -> (is, cp, vs, ρs, ras, s)
    | _ -> raise RuntimeError "Invalid state"

let rec driver = function
    | (is, cp, [v], _, _, s) when is.(cp) = IHalt -> (v, s)
    | s -> driver (step s)
```

- **Problems**:
  - Inefficient environments and lookup
  - State is now *complex* (3 separate stacks)
  - The value stack *vs* still stores complex values e.g. closures

4.3 Compiler C_2

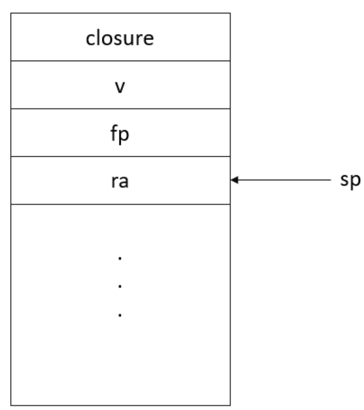
4.3.1 Activation Record

- **Problem**: Inefficient environments and *complex* state.
- **Solution**: Activation records or *stack frames*
- **Idea**:
  - Environments, return addresses and values follow *stack*-like semantics $\implies$ combine into a single stack σ.
  - The local environment and return address forms an *activation record* or *stack frame*.
- Stack frames require 2 additional pointers:
  - *stack pointer*: sp points to the top of the stack σ.
  - *frame pointer*: fp points to start of the current stack frame / activation record.
- **Problem**: Environments require a lookup $\implies$ an *addressable stack* is required.
- **Solution**:
  - Lookups must be *compiled* into offset addresses from the frame pointer fp.
  - The applied argument is placed *before* the start of the *new* stack frame
  - Local variables are placed *after* (or inside) the stack frame



Not applicable in $\lambda^{\rightarrow}$, since `let` *bindings are compiled away*

- **Problem**: Accessing free variables captured the closure ρ.
- **Solution**: Use an index into the *closure* ρ, with the closure placed *before* the start of the new stack frame (similar to arguments).



- `let` bindings and `case` branches are compiled to λ-functions using equivalences

$$\texttt{let}\ x\ :\ \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \cong (\lambda\ x\ :\ \tau.\ e_2)\ e_1$$

$$\texttt{case}\ C^i\ v_1\ \ldots\ v_{m_i}\ \texttt{of}\ (\ldots\ |\ C^i\ x_1 : \tau_1\ \ldots\ x_{m_i}\ :\ \tau_{m_i} \rightarrow e_i\ |\ \ldots)$$
$$\cong (\lambda\ x_1\ :\ \tau_1.\ \ldots\ .\ \lambda\ x_{m_i} \colon \tau_{m_i}.\ e_i)\ v_1\ \ldots\ v_{m_i}$$

```
module Instruction = struct
    type offset =
        | Stack of int
        | Closure of int

    type t =
        | ILookup of offset
        // IExtend of Indentifier.t // No-longer required
        | IMakeClosure of int * int
        | ...
        // IPopEnvironment // No-longer required
        // (due to compiled let and case branches)
        | ...

    ...
end

// scope is a mapping from Identifier.t to offsets
let rec compile scope = function
    | EPrimitive p -> pure [IPush (Primitive.interpret p)]
    | EVar x -> pure [ILookup (List.assoc x scope)]
    | EApp (e1, e2) ->
        (compile e1)
        @> (compile e2)
        @> (pure [IApp])
    | ELambda (x, _, e) -> compile_λ scope x e
    | ELet (x, t, e1, e2) -> compile scope (EApp (ELambda (x, t, e2), e1))
    | ECase (e, bs) ->
        let lbs = List.map (λ b -> (Label.fresh (), b)) bs
        and l = Label.fresh () in
            (compile e)
            @> (pure [ICase (List.map (λ (l, (c, _, _)) -> (c, l)))])
            @> (List.flat_map (λ (lc, (c, bs, e)) ->
                    let e' = List.fold_left (λ e (x, t) -> ELambda (x, t, e)) e bs in
                        (pure [ILabel lc])
                        @> (compile scope e')
                        // swaps are required since the closure for e' is TOS
                        @> (pure (List.flat_map (λ _ -> [ISwap;IApp]) bs))
```

```
                    @> (pure [IJump l]))
              lbs)
          @> (pure [ILabel l])
and compile_λ scope x e =
    let l = Label.fresh ()
    and fvs = Expr.fvs e in

    // Extend static bindings w/ x and ρ (closure)
    let x_ρ_bs =
        (x, Stack (-1))
        :: (List.mapi (λ i y -> (y, Closure i)) fvs) in
    let scope′ = List.fold_right (List.assoc) bs x_fvs_bs in

    // Compile body and lookup closure
    let (ds, is) = compile scope′ e in
    and ρ_lookups = List.map (λ y -> ILookup (List.assoc y scope)) fvs in

    // λ-definition
    let d =
        (pure [ILabel l])
        @> is
        @> (pure [IReturn]) in
            (d @ ds, ρ_lookups @ [IMakeClosure (l, List.length fvs)])
```

- **Idea**: Constant state such as instruction array *is* may be factored out

```
module Stack = struct
    type t = item array
    and item =
        | SValue of Value.t
        | SRA of int
        | SFP of int

    let show σ = ...
end

// (cp, sp, fp)
```

```
type state = int * int * int

// Stack/state helper functions
let push σ v (cp, sp, fp) =
    σ.(sp + 1) <- v;
    (cp, sp + 1, fp)
let pop (cp, sp, fp) n = (cp, sp - n, fp)
let swap σ ((cp, sp, fp) as st) =
    let i = σ.(sp) in
        σ.(sp) <- σ.(sp - 1);
        σ.(sp - 1) <- i;
        st

module Function = struct

    let apply f v σ s (cp, sp, fp) = match f with
        | FClosure (l, ρ) ->
            (l, sp, sp + 1)
            // Set up stack-frame preamble
            |> push σ (SValue (VFunction f))
            |> push σ (SValue v)
            // Set up stack-frame
            |> push σ (SFP fp)
            |> push σ (SRA cp)
        | ...

end

let lookup σ fp o = match o with
    | Stack o -> σ.(fp + o)
    | Closure o -> (match σ.(fp - 2) with
        | SValue (VFunction (FClosure (i, ρ))) -> SValue (ρ.(o))
        | _ -> raise RuntimeError "Invalid lookup")

let step is σ s (cp, sp, fp) = match is.(cp) with
    | IPush v -> push σ v (cp + 1, sp, fp)
    | ISwap -> swap σ (cp + 1, sp, fp)
    | IApp -> (match σ.(sp), σ.(sp - 1) with
```

```
    | SValue v, SValue (VFunction f) ->
        Function.apply f v σ s (cp + 1, sp - 2, fp)
    | _ -> raise RuntimeError "Invalid Stack (IApp)")
| ILookup o -> push (lookup σ fp o) (cp + 1, sp, fp)
| IMakeClosure (l, n) ->
    // pop free variables
    let ρ = Array.init n (λ i -> Value.from_stack_item σ.(sp - i))
    and st′ = pop (cp + 1, sp, fp) n in
        // push closure onto stack
        push σ (VFunction FClosure (l, ρ)) st′
| ...
| ICase cls ->
    (match matches σ.(sp) cls with
        | Some (l_c, vs_c) ->
            List.fold_left (λ st v -> push σ v st) (l_c, sp, fp) vs_c
        | None -> raise RuntimeError "Unmatched case")
| IReturn -> (match σ.(fp), σ.(fp + 1) with
    | (SFP fp′, SRA ra) -> push σ (σ.(sp)) (ra, fp - 3, fp′)
    | _ -> raise RuntimeError "Invalid Stack (IReturn)")
| ILabel l -> (cp + 1, sp, fp)
| IJump l -> (l, sp, fp)
| IHalt -> (cp, sp, fp)
```

- **Problems**:
  - The stack stores complex values (e.g. closures, sum types, pairs, etc)
  - `IMakeClosure` duplicates the values of the *free variables* of e and stores then in a closure (pushed onto the stack). Consider nested `let`s:

    ```
    let x_1 : τ_1 = e_1 in
    ...
    let x_n : τ_n = e_n in
        e
    ```

    compiles to n `IMakeClosures`, each capturing $0, 1, ..., n-1$ in their closures.

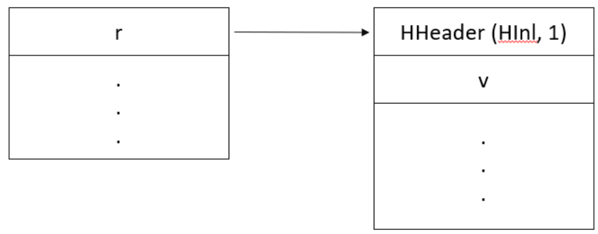
4.3.2 Heaps

- **Problem**: The stack σ still stores complex values (closures).
- **Solution**: Place complex values on the *heap*, with references to the heap stored on the *stack*.

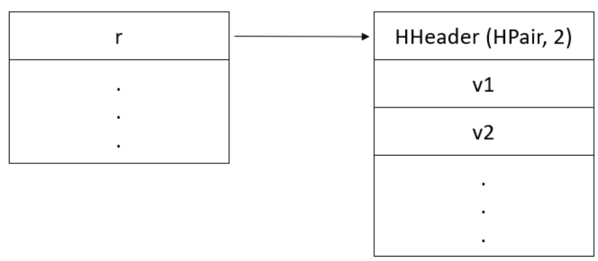
```
module Heap = struct
    type t = item array
    and item =
        | HInt of int
        | HBool of bool
        | HUnit
        | HRef of int // reference to heap
        | HLabel of Label.t
        | HHeader of int * tag // (n, t) where n is size of block
    and tag =
        | HPair
        | HInl | HInr
        | HFunction of Function.t

    // requires internal heap pointer (stored in ref)
    let alloc h n = ...
end

module Value = struct
    type t =
        | VInt of int
        | VBool of bool
        | VUnit
        | VHRef of int // reference to heap

    ...
end
```

- **Idea**: When allocating complex values (sum values, pairs, closures) of size s, allocate a block of $s + 1$ items using `Heap.alloc`, then place items from stack into heap using allocated block.

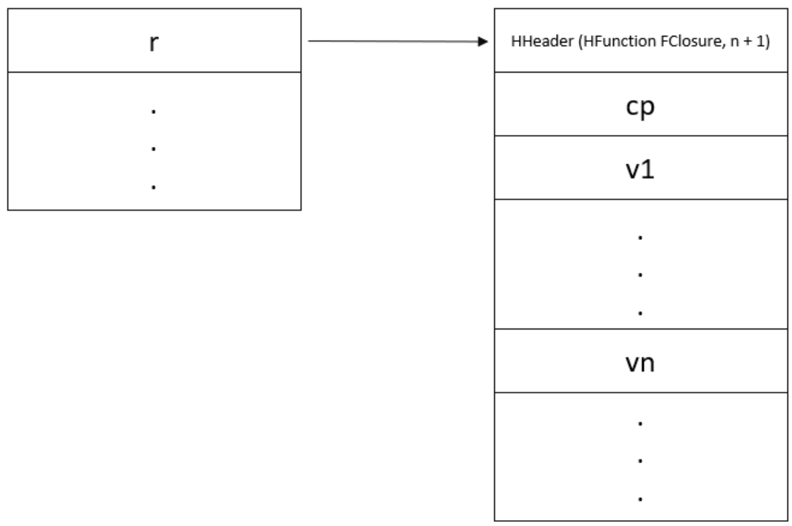
- *Store* references are simply implemented using *Heap references*.
- **Sum values**:



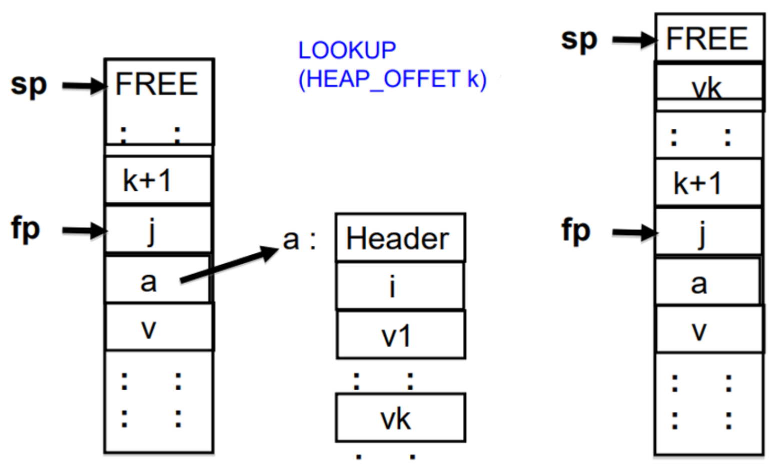
- **Pairs**:



- **Closures**:



- **Lookups**:



- **Observation**: References to the heap *prevent* the free variable value duplication implemented by closures in section ??. However, each heap lookup requires an *additional level of indirection + cost*.

```
module Function = struct
```

```
    // Modified to be tag-like, values stored on heap
    type t =
        | FClosure
        | FAdd1
        | FAdd2
        | ...

    // Reference r to heap allocated function
    let apply r v σ h (cp, sp, fp) = match h.(r) with
        | HFunction FClosure -> (match h.(r + 1) with
            | HLabel l ->
                (l, sp, sp + 1)
                |> push σ (SValue (VHRef r))
                |> push σ (SValue (v))
                |> push σ (SFP fp)
                |> push σ (SRA cp)
            | _ -> raise RuntimeError "Invalid closure")
        | ...

end

let step is σ h (cp, sp, fp) = match is.(cp) with
    | ...
    | IApp -> (match σ.(sp), σ.(sp - 1) with
        | SValue v, SValue (VHRef r) ->
            Function.apply r v σ h (cp + 1, sp - 2, fp)
        | _ -> raise RuntimeError "Invalid Stack (IApp)")
    | ...
    | IMakeClosure (l, n) ->
        let r = Heap.alloc h n in
            h.(r) <- HHeader (n + 1, HFunction FClosure);
            h.(r + 1) <- HLabel l;

            // pop free variables (convert stack items to heap items)
            let ρ = List.init n (λ i ->
                Value.from_stack_item σ.(sp - i)
                |> Heap.item_from_value)
            and st' = pop (cp + 1, sp, fp) n in
```

```
            // store heap item on heap
            List.iteri (λ i hi -> h.(r + 2 + i) <- hi) ρ

            // push closure onto stack (w/ heap reference)
            push σ (SValue (VHRef r)) st′
    | ...
```

4.4 Compiler C_3

- **Idea**: Translate C_3's bytecode into `x86` instructions. Removes stack machine $\implies$ performance improvement.

4.4.1 x86

- **x86**: A CISC register-based architecture (focused on backwards compatibility)
- Several versions:
  - 16-bit: Original version, no prefix.
  - 32-bit: Registers prefixed with `e`: e.g. `eax`
  - 64-bit: Registers prefixed with `r`: e.g. `rax`
- Registers:
  - General purpose registers: `eax, ebx, ecx, edx` (and `r8, ..., r15` for 64-bit)
  - `ebp`: Base / frame pointer
  - `esp`: Stack pointer
  - `edi`: Data pointer (points to the first applied argument)
  - `eip`: Instruction (or code) pointer
- `l, h` suffixes to denote lower and upper 8-bits of registers. e.g. `al`
- GAS syntax: Instructions has width suffixes:
  `b` (byte) = 8, `w` (word) = 16, `l` (long) = 32, `q` (quad) = 64 bits. and registers prefixed w/ `%`

`mov`	Move into register / memory. e.g. `mov %rax, %rbx` / `mov $4, offset(base)`
`lea`	Load effective address e.g. `lea offset(base), %rax`
`push,pop`	Push/pop bytes on/from stack
`not, and, or, xor`	Logical NOT, AND, OR, XOR on two registers / memory addresses, storing result in second operand.
`add, sub`	Add / subtract on two registers / memory addresses
`inc, dec`	Increment / decrement a register / memory address
`neg`	Negate a register / memory address (2's complement)
`mul, div`	Multiply / divide on two registers / memory addresses
`jmp`	Unconditional jump
`cmp, j<condition>`	Conditional jump if condition: `e, ne, z, g, ge, l, le`
`call, ret`	Call / return from a procedure

4.4.2 Registers

Definition 4.4.1. (**Stack and Register machines**) A stack-based machine (SM) is a model of computation that uses a *stack* to store values.

A register-based machine (RM) is a model of computation that uses a finite set of registers to store temporary values.

Advantages of SM	**Disadvantages of SM**
SM instructions are very compact since operands are implicit (high code density) Smaller instructions $\implies$ larger code blocks in instruction case (fewer misses) $\implies$ improved performance	Implicit operands makes optimization difficult (e.g. peephole)
SMs allow for efficient access to local variables (since offsets are computed at compile time) Spilled locals require an explicit address calculation performed at runtime e.g. `load dst sbase(offset)`.	SMs require data cache accesses for temporary values on the stack. Accessing data case is slower than registers (and may result in a cache miss) *Significant performance disadvantages*
SM ISAs reduce compiler implementation complexity. Bytecode objects generated by post-order traversals of the AST. Register machine ISAs require additional analysis e.g. register allocation. An NP-complete problem, since allocating variables to k registers $\equiv$ colouring an inference graph built by liveness analysis with k colours	
SMs have reduced states compared to RMs (registers) $\implies$ fewer required resources for implementation. Reduced state also improved interrupt mechanism, since interrupts require state to be pushed onto the stack and then jump to the interrupt handler.	

- **Problem**: Small number of registers $\implies$ allocation must be effective. (Part II Optimizing Compilers)
- **Calling conventions**: Callee and caller may share registers (for passing arguments, return results, etc) Registers are either:
  - **Caller saved**: The caller must save the value stored in the register (pushing onto the stack) if the register is used by the *caller*.
  - **Callee saved**: The callee must save the value (in the procedure preamble) in the register if the *callee* uses the register.
- **Problem**: All registers are in use
- **Solution**: *register spillage.*

Definition 4.4.2. (**Register Spillage**) If all registers are in use, then the register values are pushed onto the stack, freeing the register.

- Register spillage is also used when the number of arguments > number of data registers. Additional arguments are pushed onto the stack

5 Other

5.1 Linking and Runtimes

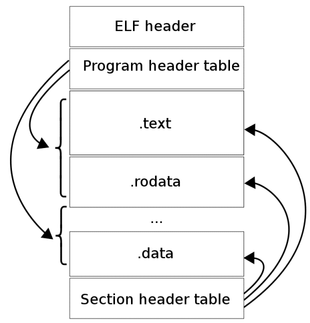
5.1.1 Static and Dynamic Linking

Definition 5.1.1. (**Object File**) An object file is the output of a compiler/assembler containing *position independent assembly/machine code*

- For each source file, compiler produces a object file.
- Compiled code but contains metadata for linking to an *executable file*.
- *Segmentation*: object files split into sections. Sections are used by *loader* and *linker*.
  - `header`: Contains descriptive information for object file w/ offsets to other segments.
  - `.text` segment: Contains the compiled assembly w/ position independent labels
  - `.data` segment: Initialized `static` / `global` variables.
  - `.rodata` segment: Initialized `const static` variables / constants
  - `.bss` segment: Uninitialized `static` variables
  - `.strtab` segment: Stores string literals
  - `.symtab` segment: Stores visibility of each declaration, declaration name (pointer to `.strtab` section), section index and virtual address (in section).

Definition 5.1.2. (**Linking**) Linking is the process of combining (*linking*) many object files to form an *executable* (binary).

- ELF (Executable and Linkable Format): standard UNIX representation of object files (and executables)



- Segmentation approach: merge individual approach (calculating linked addresses)
- Linking `.text`:
  - Object containing `main` function is placed w/ offset `0` in linked `.text` section
  - Other `.text` sections relocated w/ offsets after `main .text` section. `.symtab` is used to determine offsets (and addresses) of external labels.

Definition 5.1.3. (**Dyanmic Linking**) Dynamic linking is the process of linking object files are runtime using indirection via *DLL stubs*

Advantages	Disadvantages
Reduces executable size	Mutating dependencies. "Dependency Hell"
Updates to DLLs don't require re-linking	

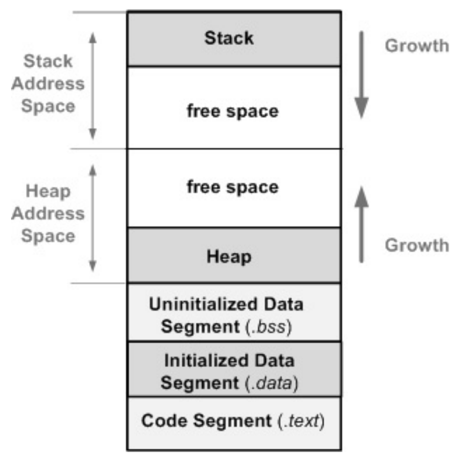
5.1.2 Runtime

Definition 5.1.4. (**Runtime Environment**) A runtime environment (or runtime system) implements the execution model of the compiled language.

- **Features**:

  - Interface w/ OS
  - FFI (Foreign Function Interfaces)
  - Type checking in dynamic languages.
  - Memory management

- A *loader* loads an executable into a process address space w/ the following layout:



5.2 Garbage Collection

- Languages w/out manual memory management via `malloc` and `free` $\implies$ runtime must implement automatic memory management, or *garbage collection*, to prevent *memory leaks*.

Definition 5.2.1. (**Heap Graph**) A heap graph is a graph $G = (V, E, R)$ where V is the set of vertices modelling nodes on the heap, edges $(u, v) \in E$ denote a pointer from u to v and $R \subseteq V$ is the set of root nodes, the set of nodes w/ pointers stores in registers / on the stack.

- A node $v \in V$ is *garbage* iff there does not exist $u \in R$ s.t $u \to^* v$.
- **Note**: Tags required for pointers vs *not pointers* (objects) $\implies$ OCaml's 63-bit integers.

5.2.1 Referencing Counting

- **Idea**: For each heap node $v \in V$, store *reference count* $\rho(v)$, the number of *pointers* to the node. We free $v \in V$ if $\rho(v) = 0$, propagating a new reference count (recursively).

```
new(v) ρ(v) ← 0;

dec(v) {
    ρ(v)--;
    if (ρ(v) = 0) {
        // traverse graph accessible from u, decrementing pointers
        for (u ∈ N(v)) dec(u);
        free(v)
    }
}

inc(v) ρ(v)++;

// assign reference (pointer) ℓ to v.
assign(ℓ, v) {
    u ← heap[ℓ];
    heap[ℓ] ← v;
    dec(u);
    inc(v);
}
```

- **Problem**: Reference counting cannot free *garbage cycles* $\implies$ memory leaks and space overhead to maintain count $\rho(v)$.

5.2.2 Mark and Sweep

- **Idea**: Traverse the heap graph G from the root set R, marking accessible nodes $v \in V$, denoted $\rho(v) \geq 1$. Iterate over V, freeing vertices $v \in V$ w/ $\rho(v) = 0$ (inaccessible).

```
new(v) ρ(v) ← 0;

mark(G) {
    W ← R; // dfs/bfs from root set
    while W ≠ ∅ {
        w ← pop(W);
        ρ(w)++;
        if (ρ(w) = 1) { // first visit
            W ← W ∪ N(w);
        }
    }
}

sweep(G) {
    for (v ∈ V) {
        if (ρ(v) = 0) free(v);
        ρ(v) ← 0; // clear count
    }
}

collect(G) {
    mark(G);
    sweep(G);
}
```

- Computes the reflexive transitive closure of R (on the relation defined by G), denoted R^*, the set of objects reachable from the root set R.

  The unreachable objects is given by $V_F = V \backslash R^* = \{v \in V : \rho(v) = 0 \text{ after mark}(G)\}$

	Reference Counting	Mark and Sweep
Collection	Incremental	Batch
Cost Per Assignment	High	Low
Delays	Short	Long
Collects Cycles	No	Yes

5.2.3 Copy and Generational Collectors

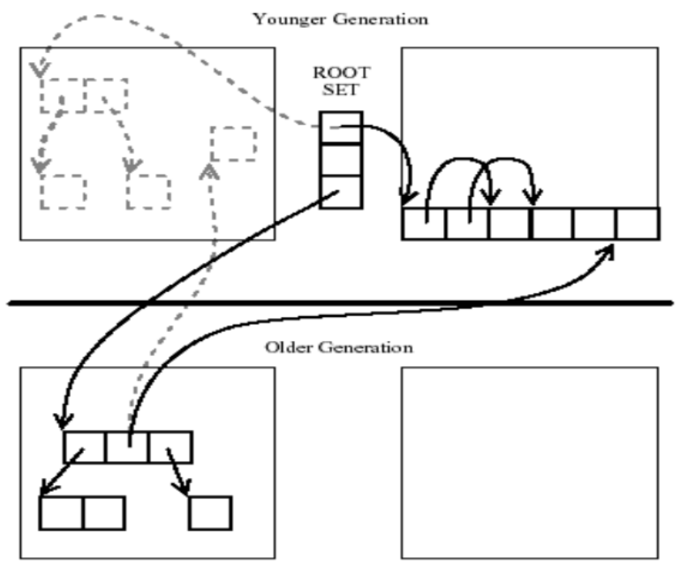
- **Idea**: Idea, remove second iteration of Mark and Sweep using 2 heaps (Time $\to$ Space tradeoff) $\implies$ *Copy collector*.

```
// 2 heaps H_1, H_2 w/ G representing current heap

new(v) ρ(v) ← 0;

mark(G) {
    W ← R; // dfs/bfs from root set
    while W ≠ ∅ {
        w ← pop(W);
        ρ(w)++;
        if (ρ(w) = 1) { // first visit
            move(w); // moves object w to next heap
            W ← W ∪ N(w);
        }
    }
}


collect(G) {
    mark(G);
    swap(); // swaps current heap w/ next heap via pointer swap
    clear(); // clears next heap
}
```

- **Advantage**:
  - Reduces # of traversals over G (runtime now proportional to live objects)
  - `move` function may implement compaction $\implies$ eliminates external fragmentation

  Disadvantages:
  - $\times 2$ amount of memory used at a given time.

- **Observation**: Approximately 95% of objects are freed quickly $\implies$ inefficient to continue copying long-lived objects between heaps H_1, H_2.
- **Idea**: Split H_1 and H_2 into *generational heaps*, aging objects from younger generations into older ones if they survive a collection. Older generations are collected less frequently than younger ones:



A *generational copy collector*.

5.3 Static Links

- **Problem**: Closures are slow and inefficient (*especially Timothy G's*)

5.3.1 Escaping Variables

- A scope S, denoted S_f, is a set of available variables associated in some λ abstraction (denoted $f = \lambda x.e$).

Definition 5.3.1. (**Lexical Scoping**) Lexical scoping is where the scope S is defined lexically (statically).

Dynamic scoping is where the scope S is defined dynamically by the stack frames.

- A scope S_f is said to be the *definer or binder* of $x \in V$ iff x is defined in f.
- Scopes are *nested* / recursive:

```
let twice f =
    let g x = f (f x) in g
```

  Scope S_g is nested in S_{twice} w/ nesting depths d and $d + 1$.

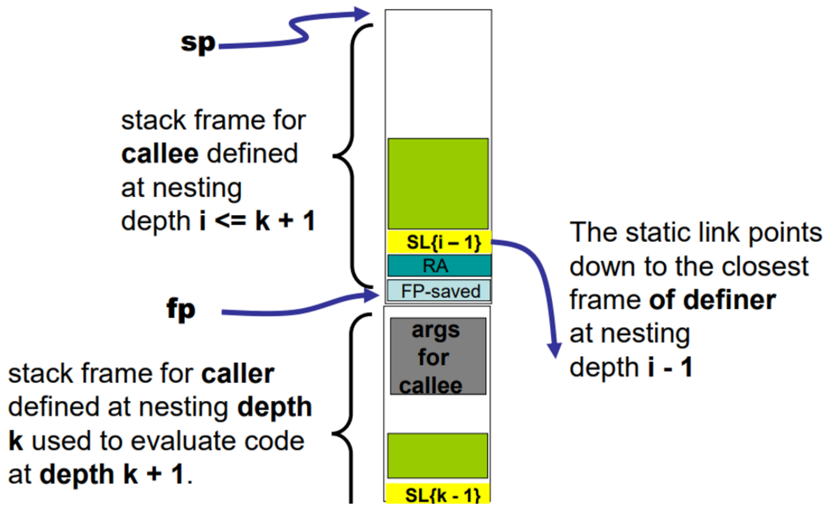
Definition 5.3.2. (**Escaping Variable**) A variable x is *escaping* wrt S_f if x is free in some function g that *escapes* f (returned by f)

- $\implies$ Escaping variables must be allocated on the *heap* once *closed over* (deallocated by current stack frame). An *escaping-variable record* is placed in the stack frame (like a *closure*).
- **Escape analysis**:
  1. Function (w/ closure) is returned
  2. Function is assigned to a global variable / reference
  3. Function is stored in a data structure that escapes
  4. Higher order functions.

  See Ali compiler for upvalue analysis*

5.3.2 Static Link Chains

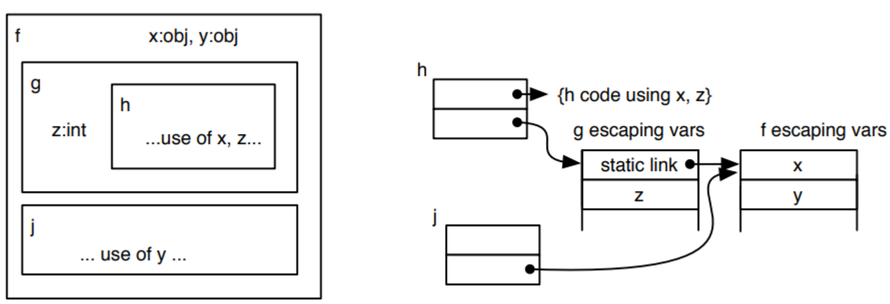
Definition 5.3.3. (**Static Link**) A pointer (in the stack frame) to the definer (or enclosing scope) of the current function.



- Static linking implements the *nested* scope traversal (defined by lexical scope) via *static links*.
- Accessing variables x w/ index i and static distance d:

```
do {
    frame ← !frame.static_link;
} while (--d > 0);
return frame.vs[i];
```

- With closures, static links may be pointed to *closures*:



5.4 Optimizations

5.4.1 Inlining

Definition 5.4.1. (**Inlining**) Inline expression is an optimization technique by inlining the function body for a given call.

- Optimizes running time by eliminating overhead of call prologue and epilogue . Space $\rightarrow$ Time tradeoff by duplicating function bodies.
- **Heuristics** for Inline expansion:
  1. Expand function-calls that are executed frequently. Determines by static-analysis of loop-nest depth (or at runtime w/ JIT)
  2. Expand functions w/ very small bodies. Minimizing code explosion from inline expansion.
  3. Expand functions that are only called once (and aren't exported), then apply dead-code elimination (deleting original function definition).

5.4.2 Constant Folding

Definition 5.4.2. (**Constant Folding**) Constant folding is an optimization where constant expressions are evaluated at compile time.

- Optimizes running time. Compile time $\rightarrow$ Running time tradeoff.

Definition 5.4.3. (**Constant Propagation**) Constant propagation is the process of substituting the identifier x of a constant expressions e evaluated at compile time w/ it's value v.

- Implementing using *reaching definition analysis* (binder dependencies).

Advantages	Disadvantages
Eliminates the instructions required by the calling convention: the prologue and epilogue (and the `call`/`jump` required).	Inlining duplicates function body, increase in code size $\implies$ may result in a *code explosion*.
Reduces register spillage from arguments	Increases *working set* (the set of pages the program requires access to at a given time). Increases page faults $\implies$ thrashing (in serve cases)
Removes callee, caller boundary, allowing for more intra-procedural optimizations (e.g. peephole, etc)	Increases instruction cache miss rate (due to increased code size).
Eliminating function calls improves the temporal + spatial locality of instructions.	

Original	Optimized

`int x = 5;`
`int y = x * 2;`
`int z = a[y];`
	`int z = a[10];`

Advantages	Disadvantages
Increases runtime performance	Decreases compiler performance (massive slowdown for large programs)
Unsafe optimizations may be applied. e.g. `0 *` $x \neq$ `0` (by IEEE)	
Simple implementation. Pattern matching on operators w/ operands tagged `Const`.	

5.4.3 Peephole Optimizations

Definition 5.4.4. (**Peephole Optimization**) Peephole optimization is an optimization technique applied to a small set of instructions (*known as a peephole or window*), where the small set of instructions is optimized to an equivalent set of instructions w/ better performance

- May be applied to ISA or expressions (low and high level) w/ techniques:
  - Removing null sequence of operations
  - Combining operations into a single instruction
  - Algebraic laws
  - Re-ordering instructions, such as load hoisting or loop invariant optimizations, etc.

Advantages	Disadvantages
Many applicable optimizations	Limited scope due to *peephole*. Cannot deal w/ optimizations based on control flow / inter-procedural optimizations
Simple implementation. Pattern matching optimizations w/in peephole optimization w/ traversal algorithm across AST / linearized code.	
Combined with inter-procedural optimizations (e.g. inlining) yields extremely effective optimizer.	