1 Parsing

Lexing converts input string to *tokens* with a *regex* (matches literals, alternation, sequencing & iteration), *context free grammar* (regex + recursion) or *context sensitive grammar* (CFG + arbitrary state).

A token may have multiple meanings, or can be a prefix to another token (e.g. ≫). *Tokenizers dont have enough context.* Instead, scannerless parsing operates on character streams.

1.1 Context Free Grammar

There are two different types of CFG:

	$LL(k)$	$LR(k)$
Approach	Top Down	Bottom Up
Left Rec.	No	Prefers It
Good Errors	Can be	Weak
Ambiguity	Up to k tokens.	
Complexity	$\mathcal{O}(n)$; $\mathcal{O}(1)$ choices	$\mathcal{O}(n)$

Any CFG can be passed in $\mathcal{O}(n^3)$, and fully unambiguous with multiple parses (CYK algorithm). $LL(k)$ and $LR(k)$ have ambiguities that can be resolved by looking ahead k tokens. Aim for $k = 1$ languages.

A grammar is $LL(1)$ if $\forall$ rules $A \to \alpha \mid \beta$:
- $\texttt{fst}(\alpha) \cap \texttt{fst}(\beta) = \emptyset$
- $\epsilon \in \texttt{fst}(\alpha) \Rightarrow [\texttt{fst}(\beta) \cap \texttt{follow}(A) = \emptyset]$
- $\epsilon \in \texttt{fst}(\beta) \Rightarrow [\texttt{fst}(\alpha) \cap \texttt{follow}(A) = \emptyset]$

1.2 Parsing Expression Grammar

A **PEG** is unambiguous by construction. Can parse any $LR(k)$ grammar, not necesarilly any CFG. Returns a single parse tree in $\mathcal{O}(n)$ time with no backtracking, $\mathcal{O}(2^n)$ time with backtracking on every decision. It consists of: *literals* (`'x'`), *variables* (`v`), *empty* (ϵ), *sequencing* (`e1 e2`), *left biased alternation* (`e1 / e2`), *negative lookahead* (`!e`), *grouping* (`e`) and *end of file* (`eof`). Additionally, we have redundant *any char* (`.`), *classes* (`[0-9]`), *optional* (`e?`), ≥ 0 (`e*`), > 0 (`e+`) and *lookahead* (`&e`).

PEGs resolve ambiguity with *left biased alternation*. They are also **greedy** - they consume as much as they can. More ambiguous branches should be to the left:

```
stmt <- "if" expr "then" stmt "else" stmt
   / "if" expr "then" stmt
```

Do not use *left recursive* or *nullable* rules - cause infinite loops in PEGs. To reduce backtracing while preserving meaning we **left-factor** `e e1 / e e2 => e (e1 / e2)`. Do *NOT* `e1 e / e2 e => (e1 / e2) e`. We resolve left recursion with: `E <- E OP / T => E <- T OP*`.

Renaming: Optional phase to distinguish vars of the same name, ensuring each var is decl before use.

2 Semantic Checking

Typechecking may be **bottom-up** (return the type of child exprs and compare at every level) or **top-down** (push info about expected types down the tree, check at leaves). *We want to parse not validate.* It makes sure types are sound.

Scopechecking makes sure vars are defined and unique.

```
enum Constraint {
  case Is(refTy: Type)
  case IsNumeric
  case IsRecord
}
object Constraint {
  val Unconstrained = Is(?)
  val IsArray = Is(KnownType.Array(?))
}

// Check if type matches constraint:
extension (t1: Type) def ~(t2: Type): Option[Type]
  = (t1, t2) match {
    case (?, t2)                => Some(t2)
    case (t1, ?)                => Some(t1)
    case (Array(t1), Array(t2)) => t1 ~ t2
    case (t1, t2) if t1 == t2   => Some(t1)
    case _                      => None
  }
extension (t: Type) def satisfies(c: Constraint)(
      using ctx: Ctx[?]): Option[Type]
 = (t, c) match {
  case (t1, Is(t2)) => (t1 ~ t2).orElse {
    ctx.error(Error.TypeMismatch(t1,t2))
  }
  case (?, _) => Some(?)
  case (t@(T.Int | T.Float), IsNumeric) => Some(t)
  case (t, IsNumeric) =>
    ctx.error(Error.NonNumericType(t))
  case (t@T.Record(_), IsRecord) => Some(t)
  case (t, IsRecord) =>
    ctx.error(Error.NonRecordType(t))
}

// Ctx Definition:
class Ctx[C](i: TypeInfo, errs: mutable.Builder[
      Error, C]) {
  def errors: C = errs.result()
  def typeOf(v: String): KnownType = i.var(v)
  def typeOf(r: String, f: String):
    Option[KnownType] =i.rec(f).get(f)
  def error(err: Error) = errs += err
}

// Actually check, for example:
def check(e: Expr, c: Constraint)(using Ctx[?]): (
      Option[Type], TypedExpr) = e match {
  case Expr.Add(x, y) => checkNum(x, y, c)(
      TypedExpr.Add.apply)
  case Expr.Num(n) => (T.Int.satisfies(c),
      TypedExpr.Num(n))
}

def checkNum(x: Expr, y: Expr, c: Constraint)
(build: (TypedExpr, TypedExpr, Type) => TypedExpr)
(using Ctx[?]): (Option[Type], TypedExpr) =
  val (lt, xTyped) = check(x, IsNumeric)
  val (rt, yTyped) =
    check(y, lt.fold(IsNumeric)(Is(_)))
  val ty = best(lt, rt)
  (ty.satisfies(c), build(xTyped, yTyped, ty))

def best(t1: Option[Type], t2: Option[Type]): Type
      = (t1, t2) match {
  case (Some(?), Some(t)) => t
  case (Some(t), _)       => t
  case (None, t)          => t.getOrElse(?)
}
```

3 Basic Codegen

```
genStmt (Asgn id e) = genExpr e ++ [Pop id]
genStmt (Seq s1 s2) = genStmt s1 ++ genStmt s2
genStmt (For id e1 e2 body) =
  genExpr e1 ++ [Pop id] ++ -- Init loop var
  [Label l1] ++ -- Define loop start
  -- Check the loop condition:
  genExpr e2 ++ [Push id, CmpGt, JTrue l2] ++
  genStmt body ++ -- Loop body
  [Push id, Push 1, Add, Pop id] ++ -- Inc counter
  [Jmp l1, Label l2] -- Jump back to loop start

genExpr (Binop op e1 e2) = genExpr e1 ++ genExpr e2
      ++ genOp op
genExpr (Unop op e) = genExpr e ++ genOp op
genExpr (Ident id) = [Push id]
genExpr (Const n) = [Push n]
```

3.1 Using Registers

```
genExpr (Const n) r = [LoadImm r n]
genExpr (Ident i) r = [Load r i]
genExpr (BinOp op e1 e2) r = genExpr e1 r ++
      genExpr e2 (r + 1) ++ genOp op r (r + 1)

-- We also allow special cases for imm operands
genExpr (BinOp op e (Const n)) r = genExpr e r
  ++ genOpImm op r n -- e.g. [AddImm r n]
genExpr (Add (Const n) e) r
  | commutative op = genExpr e r ++ genOpImm op r n

-- If we run out of regs we can push onto stack:
genExpr (Binop op e1 e2) r
  | r == MAXREG = genExpr e2 r ++ [Push r] ++
      genExpr e1 r ++ genOpStack op r
  | otherwise = genExpr e1 r ++ genExpr e2 (r + 1)
      ++ genOp op r (r + 1)
```

3.2 Sethi-Ullman Algorithm

Let $A \circ B$ be an operation, where expr A requires a regs and B requires b regs. If A is evaled first, max regs used is $\max(L, R+1)$; otherwise its $\max(L+1, R)$.

```
weight (Const _) = 1
weight (Ident _) = 1
weight (Binop _ e1 e2) = min [c1, c2]
  where c1 = max (weight e1) (weight e2 + 1)
        c2 = max (weight e1 + 1) (weight e2)

-- For commutative ops
genExpr (Binop op e1 e2) r =
  if weight e1 > weight e2
    then genExpr e1 r ++ genExpr e2 (r + 1) ++
      genOp op r (r + 1)
    else genExpr e2 r ++ genExpr e1 (r + 1) ++
      genOp op r (r + 1)

-- For non-commutative ops
genExpr (Binop op e1 e2) (r:r':rs) =
  if weight e1 > weight e2
    then genExpr e1 (r:r':rs) ++ genExpr e2 (r':rs)
      ++ genOp op r r'
    else genExpr e2 (r':r:rs) ++ genExpr e1 (r:rs)
      ++ genOp op r r'
```

The Sethi-Ullman algorithm doesn't have context of the vars or exprs around it, so it cannot keep regs to store the same var.

3.3 Graph Colouring

Reg alloc has a very big impact on performance. We can build a smart allocator with **graph colouring**:

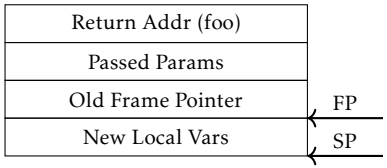
1. A *tree-walking translator* makes intermediate code where temp values are saved in named location.
2. Make an **interference graph**: nodes are temp locations, linked by an arc if the values must be stored simultaneously (their *live ranges* overlap).
3. Try to colour the nodes, obtaining a reg allocation.

Although this is slow, a fast heuristic can be used. If this fails, we must **spill** into memory. For efficiency, *prioritise nesting depth* when spilling (dont spill innermost loop); perform a split that *enables colouring*.

3.4 Function Calls

To make sure caller args end up in the correct regs, we can use **register targeting**. To ensure regs dont get overwritten, we have **caller-saved** and **callee-saved** regs. Since neither knows info about the other, we must save ALL regs when calling.

If a variable persists after a function call, then saving in *caller saved* avoids clobbering value. The structure of the **stack** when a function is called:



4 Optimisation

Peephole optimisation replaces obviously inane assembly code patterns with more efficient ones. There are infinite possibilities, but *phase ordering problem* - which order to we apply optimisations in.

4.1 Loop Optimisation

- **Loop Invariant** instructions (operands outside the loop) can be moved out of the loop.
- **Strength Reduction** replaces *induction variable* (value changes by loop invariant amount each iteration) with a simpler expression. You can repr mulitple ind vars with 1 reg if they have a linear relationship (1 in terms of other).
- **Control Var Selection** replaces loop control variable (e.g. `i`) with an induction var used in the loop.

4.2 Loop Optimisation - Finding an Invariant

A node (instr) defines a variable if it assigns a value to it. It uses a variable if it reads its value. A definition is *loop invariant* iff **all the defs** of each operand come from outside the loop *or* theres **only one *reaching* def** and that def is loop invariant. *An instruction is loop-invariant only if the things it uses are also loop-invariant or fixed outside the loop.*

Reaching def: the definition of var `t` reaches a point `p` if theres a path from the def to `p` with no *intervening redefinition* of `t`. It is ***relevant*** iff it is also used in the instr.

$$\texttt{ReachIn}[n] = \bigcup_{p \in \text{pred}(n)} \texttt{ReachOut}[p]$$

$$\texttt{ReachOut}[n] = \texttt{Gen}[n] \cup (\texttt{ReachIn}[n] - \texttt{Kill}[n])$$

Where `Gen` is the def generated at n and `Kill` is the defs that are invalidated at n. To solve, start with all sets $\emptyset$, it. until stable (*monotonic* & *converging* process). We can use this to find loop-invariant instrs.

4.3 Loop Optimisation - Finding Loop Header

To place it correctly we need to find the loop header. Node `d` **dominates** `n` if every path from the start node to `n` must pass through `d`. ***Every node dominates itself***, and the start node dominates every other node. To find dominators:

1. $\texttt{Doms}[s] = \{s\}$ for the start node.
2. $\texttt{Doms}[n] = \xi$ - all nodes is most conservative guess.
3. Iterate $\texttt{Doms}[n] = \{n\} \cup (\bigcap_{p \in \text{pred}(n)} \texttt{Doms}[p])$. This converges quicky due to *monotonic shrinkage*.

A **back edge** is an edge from n to h s.t. h *dominates* n: represents a *loop*. The natural loop is set S containing header h such that:

- $\forall s \in S, \exists$ a path from s to h.
- $\forall s \in S, \exists$ a path from h to s.
- $\forall s \notin S, \nexists$ a path from s to any of $S \setminus \{h\}$.

These may correspond to a single source-code loop *(e.g., multiple exits or re-entry points)*.

If loop B has header b and lies entirely within loop A with header a and $b \in A$ then B is **nested** in A. A **loop tree** contains each loop as a node.

Since a loop header can have *multiple predecessors*, insert a new block (**pre-header**) that has a single edge to the loop header, is its only predecessor. This is a save place to hoist invariant code.

4.4 Loop Optimisation: Hoist Safety

Not all *loop invariant* expressions can be hoisted:

- Instruction must **dominate all loop exists**.
- One def of the var exists in the loop (**uniqueness**).
- The var is **not live out** from the pre-header.

Single Static Asgnment (SSA) ensures each var is asgned once by splitting overlapping live ranges. This removes the need to check for uniqueness and liveness. At **control flow joins** (i.e. after an if-else), we insert a dummy **phi-node** $a_3 = \varphi(a_1, a_2)$ which magically picks a_1 or a_2 depending on which path is taken. This is later eliminated during codegen.

4.5 Data Flow Analysis

Each node in a **control flow graph** stores its *using regs*, *defs regs* and *succsessor ids*. We define:

- `IN[n]`: Set of live regs immediately *before* n: live after n and not overwritten by n, or it is used by n.
- `OUT[n]`: Set of live regs immediately *after* node n: it is live before any of n's successors.

$$\texttt{OUT}[n] = \bigcup_{s \in \text{succ}(n)} \texttt{IN}[s]$$

$$\texttt{IN}[n] = \texttt{uses}[n] \cup (\texttt{OUT}[n] - \texttt{defs}[n])$$

To find all live ranges, it. until fixed point is reached:

```
for (Node n : cfg) {
  Var[] IN = [];
  Var[] OUT = [];
}
do {
  for (Node n : cfg) {
    Var[] oldIN = IN[n];
    Var[] oldOUT = OUT[n];
    IN[n] = uses[n] + (OUT[n] - defs[n]);
    OUT[n] = succ(n).map(IN[_]).reduce(_ + _);
  }
} while (IN[n] != oldIN || OUT[n] != oldOUT);
```

From the live ranges we derive *interference graph*, then colour and update the reg allocation.

4.6 Loop Scheduling

Loop Scheduling Optimisation reorders how loops are executed to use *vector instructions*, *multiple cores* and improve *cache utilisation*.

4.7 Loop Scheduling: Dependence Analysis

To use vector instrs, we must verify iterations are truly parallel - we must find **loop carried dependence** (whether each iteration depends on the previous). To detect this:

1. Let $\texttt{IN}[S]$ be the set of mem locs read by S.
2. Let $\texttt{OUT}[S]$ be the set of mem locs written to by S.

Reordering is constrained by 4 types of dependence:

- **Data** dependence: $S_1 \delta\ S_2$ or $\texttt{OUT}[S_1] \cap \texttt{IN}[S_2]$. *$S_1$ must write something before S_2 can read it.*
- **Anti** dependence: $S_1 \overline{\delta}\ S_2$ or $\texttt{IN}[S_1] \cap \texttt{OUT}[S_2]$. *$S_1$ must read something before S_2 overwrites it.*
- **Output** dependence: $S_1 \delta^o S_2$ or $\texttt{OUT}[S_1] \cap \texttt{OUT}[S_2]$. *If S_1, S_2 write to a loc, S_1 must write first.*
- **Control** dependence: $S_1 \delta^c S_2$. *S_1 determines whether S_2 should execute.*

Consider two iterations I_1, I_2. A dependence occurs between statements S_p, S_q if S_p in I_1 references the same mem loc as S_q in I_2. This may occur if they both refer to a common array A for some subscript expr φ. If $S_p = A[\varphi_p(I)]$ and $S_q = A[\varphi_q(I)]$ then a dependence occurs iff $\phi_p(I_1) = \phi_q(I_2)$ for integer I_1, I_2 in the loop bounds.

We have a *data dependence* if:

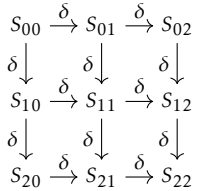
- $A[\varphi_p(I)] \in \texttt{OUT}[S_p]$ and $A[\varphi_q(I)] \in \texttt{IN}[S_q]$.
- The asgns precede uses: $\forall I_1, I_2 . I_1 < I_2 \Rightarrow S_p \delta_< S_q$.

An *anti dependence* if the uses precede the asgns: $\forall I_1, I_2 . I_1 > I_2 \Rightarrow S_p \overline{\delta_<} S_q$. If $\exists I_1, I_2 . I_1 < I_2 \wedge \exists I_1, I_2 . I_1 > I_2$ then $S_p \delta_* S_q$: we *must respect execution ordering*, but can't classify dependency. If $\forall I_1, I_2 . I_1 = I_2$ there are no dependencies *within* iteration of the loop, but no loop-carried dependencies: $S_p \delta_= S_q$.

If $\forall I_1, I_2 . I_2 - I_1 = k$ then k is the **dependance distance**. When optimising for *cache performance*, consider *reuse relationship* $\texttt{IN}[S_1] \cap \texttt{IN}[S_2]$. There is no dependence, but cache performance is faster for smaller reuse dist.

4.8 Loop Scheduling: Nested Loops

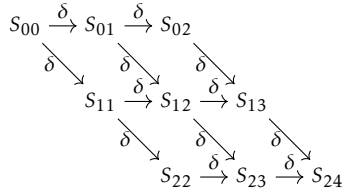
To show iteration-to-iteration dependencies in a nested loop we use an *iteration space graph*:



In this example, the inner loop is not vectorisable since there is a dependence chain linking successive iterations. Similarly, outer loop is not parallel. This loop nest has two dependence distance vecs, $(1,0)$ by outer loop and $(0,1)$ by inner loop.

This loop is **interchangable** - we can change which loop is inner and outer - but this does not improve vectorisability or parallelisability. An interchange is *invalid* if $\exists$ dependence distance vec (i,j) s.t. $i,j > 0$.

One transformation we can do is skewing the computation:



Now we can interchange for **top down lexicographic traversal** (i.e. $S_{00}, S_{01}, S_{11}, S_{02}, S_{12}, S_{22}, \cdots$). Now, iterations in each column are independent, so the inner loop is vectorisable.

A loop nest can be interchanged if the *transposed* dependence dist vecs are *lexicographically forward*. Skewing is *always valid*, exposing parallelism by aligning parallel iterations with one of the loops. Skewing can make interchange valid.

5 Runtime Organisation

- **Primitives** require diff mem amounts. Optimal access aligns vars to boundaries / PCIE bandwidth.
- **Records** are key-val pairs. Can be different sizes, but allocated *consecutively* in mem for easy access and efficiency.
- **Arrays** are groups of vars of the same type. Alloced conescutively, access with base addr & offset.
- **Objects** referenced by ptr, contain ptr to *method lookup table* and raw vals for data fields (access with addr offset). To call a method, pass object reference as a *hidden parameter*. For inheritance and overriding, a new method lookup table is created, with a ptr to the parent's method lookup table. This allows for **dynamic dispatch** (selecting impl of a polymorphic operation). **Dynamic binding** can also be done by copying an obj ref.

Local vars persist for the duration of the method call. Located in the **stack seg** of the *program addr space*. Access with a **frame ptr register** (pointing to stack frame base) and an offset.

Global vars persist for the duration of the program. Located in the **data seg (static)** of the *program addr space*. Access with **base mem addr**, no frame ptr required.

Dynamic vars persist until garbage collection. Located in the **data set (heap)** of the *program addr space*.

6 Heap Management

- **Heap Allocation** maps dynamic vars in the heap.
- **Heap Deallocation** frees inactive mem space.
- **Heap Compaction** improves mem utilisation and efficiency while removing fragmentation.

Heap is managed in **blocks**, containing *housekeeping data* (e.g. size, status) and *object data* (fields, methods). Object references point to the object data, not the start of the block. Traditional alloc done with **free list** of free blocks in the heap. We do:

1. If a free block of exactly the right size exists, return it.
2. If a bigger block exists, split into (a) of right size and (b) of remaining size; return (a).
3. If no block found, request mem from OS.

This is slow. Instead, maintain *many free lists* for different block sizes, or allocate more space for *data that may grow*. We need to consider *memory alignment* and the *negative impact on caching* when we dont have *spacial locality*.

6.1 Garbage Collection

GC dynamically deallocs from the heap by one of:

- **Reference Counting** - each block keeps refcount in *housekeeping* data. If copied, incremented. If deleted, decremented by num of loc vars that ref them. When the count reaches zero, deallocate mem block. Cascade to any obj it points to. *In total the number of objs pointing into it.* Extra code required for pointer manipulation, and cannot handle cyclic refs.
  - *When:* Immediately on count reaching zero.
  - *Perf:* Low overhead per operation, good for short-lived objects. Fails with cycles, and overhead grows with pointer-heavy code.
- **Mark & Sweep**: *Phase 1 marks* blocks as live that are reachable from non-heap references. *Phase 2 sweeps* through heap, deallocating unmarked blocks. Slower than refcount, but handles cyclic refs. Also batches deallocs for efficiency.
  - *When:* Triggered when heap usage crosses threshold or periodically.
  - *Perf:* Pause-the-world; can cause latency spikes. Better throughput than refcounting, especially with cycles.
- **Pointer Reversal** avoids using extra mem when traversing blocks. When moving from child to parent, *overwrite* child pointer back to its parent. After finishing, backtrack to the parent and *restore* the original pointer. Faster than mark & sweep but requires more memory.
  - *When:* During mark phase of mark & sweep.
  - *Perf:* Reduces stack space (no recursion), but incurs pointer rewrites. Mostly theoretical or used in constrained systems.
- **Two Space**: Split heap into *from* space and *to* space. When from space full, copy all live blocks to to space. No pointer manipulation, better spacial locality, not efficient for large heaps.
  - *When:* When *from* space fills up.
  - *Perf:* Fast allocation (bump pointer), fast collection (copying). Wastes 50% of heap. Poor fit for long-lived or large objects.
- **Generational** - divide heap into areas based on block age. Adaptively perform different GC algorithms for different areas.
  - *When:* Young gen collected frequently, old gen rarely.
  - *Perf:* Optimized for the weak generational hypothesis (most objects die young). Reduces pause time, excellent for interactive apps. Complexity in implementation.