Compiler Design

COL 728 Lecture Notes

Basics

- Compilers vs Interpreters, and partial evaluation

  Interpreters

  - Like a function, which maps code(program) and data(input to the program) to output

-
  - "Online" evaluation
  - Interpret : (Program, Data) → Output
  - Interpret : Program → Data → Output
  - Interpret : Program → (Data → Output)
  - Interpret : Program → Executable

  Compilers

  - "Offline evaluation"
    - Compile : Program → Executable

    Partial Evaluation

    - A program can be seen as a mapping of input data to output data
    - Prog : (I_{static} x $I_{dynamic}$) → Output
    - Residual Program, possibly more efficient, $Prog_*$: $I_{dynamic}$ → Output
    - Specialiser : (Prog, I_{static}) → $Prog_*$ a partial evaluator

---

Futamura Projections

- Describe when "Prog" is an interpreter, i.e. if I_{static} is source code, partial evaluation to a "$Prog_*$" gives a version of the interpreter running only that code
- Essentially, $I_{dynamic}$ is neglected
- This "specialised" version of the interpreter is written in the language in which the interpreter has been written (e.g. Assembly for JAVA)
- "$Prog_*$" is effectively the compiled version of I_{static}

The projections

1. Specialising an interpreter for given source code, yields an executable
   - Interpret : Prog → Exec
   - Specialising Interpret for prog, Spec1 : (Interpret, Prog) → Exec

    - Spec1 : Interpret → (Prog → Exec) [= Compile]
2. Specialising the specialiser for the interpreter, yields a compiler
    - Spec1 : Interpret → Compile
    - Spec2 : (Spec1, Interpret) → Compile
    - Spec2 : Spec1 → (Interpret → Compile)
3. Specialising the specialiser for itself, yields a tool that can convert any interpreter to an equivalent compiler
    - Spec2 : Spec1 → InterpretToCompileConverter

Take-aways

- Executable is the partial-evaluation of an interpreter for a program's source code, generated during *compilation*
- Execution of the executable should ideally be faster than the execution of the interpreter on the original program, and this speed-up determines the "quality of compilation". A compiler must aim to optimise in both *space* and *time*.
    - A trivial compilation stores the interpreter and source code as a tuple, so on execution the entire thing runs again, leading to worse run-time
    - Smarter compilation involves substitution of I_{static} into the interpreter's implementation, which may involve things like constant-propagation, loop-unrolling, local optimisations for static statements, etc
    - Caching Generated code (very effective for loops), to avoid regenerating the same machine code many times
        - Static Compilation - Ahead-Of-Time - cache translation
            - *Offline*, no compilation cost at run time
            - *Expensive*, given a time budget, can't decide which part to optimise more (don't know which region of code is "hot")
            - Compile + Exec may be slower than plain interpreted (*no loops*)
        - Dynamic Compilation - *Just-In-Time* - machine code generated at run time, while caching translations for each program segment
            - Can spend more effort on optimising "hot" regions
            - If generated code is larger than program, more *space efficient* (FB 250MB vs. 90MB)
            - Pay compilation *cost at run time* - not too bad for small app running long (due to loops), can also benefit from multiple cores
    - Global optimisations - spanning multiple code segments due to
        - Sub-optimal code like `x = 1; x = 2`
        - Machine representation being richer than source code syntax (complex opcodes)
        - Optimisations leading to others (one substitution followed by many more). *Caveat* - optimisation can preclude others (e.g. *shift* precludes *multiply-and-add*)
        - Generating an optimal implementation for a given program specification in Turing's model is undecidable, and NP-hard for the finite model

---

Why Compilers?

- Moore's Law running out of steam as transistor scaling approaches physical limits - source-to-drain leakage, limited gate metals/channel materials
- Process and materials engineers' innovations meant that architecture and systems was always playing catch-up, the OS and programming stacks of today largely similar to those in the 60's
- Many innovative ideas, haven't managed mainstream adoption, as we can't map human-written programs to these efficient architectural structures, and coding directly with these in mind is difficult/non-intuitive
- Compiler designs of today are reminiscent of yore, which handled simple opcodes, and have features like vectorisation, parallel constructs, etc. tacked on, leading them to become very large and complex pieces of code
- Program complexity has drastically increased, from calculators to NLP, leading to programming in higher levels of abstraction - objects and automatic memory management, MapReduce and functional languages, ML constructs like neural networks
- Higher levels of abstraction increase the gap between the programming language and the architectural abstractions, making the search space for optimal implementations even larger
- Domain-specific languages like ForTran for scientific computing(support for FLOPs, arrays, parallelism), SQL for business apps(persistence, report generation), C/C++ for systems programming(fine-grained control over resources), HTML for web dev, etc. all of which require different types of optimisation support. Building a general framework that covers all domains, yet provides the best optimisation is very hard problem.

ForTran - Birth and Basic Design

- 1954 : IBM develops the 704, the first commercially-successful machine. It found that software costs greatly exceeded the hardware costs
- 1953 : John Backus introduces "Speed coding" - a small interpreted language much easier to program in than directly programming the machine. Didn't catch on as -
    - Much slower, 10 - 20x
    - Interpreter itself took 300 bytes of memory (which was 30% of all available memory)
- Leveraging this experience, and the fact that most computation was of scientific formulae, Backus developed a higher-level programming language to translate formulae into machine code, called it "ForTran"
- Developed from 1954 to 1957; by 1958, 50% of all programs were in ForTran1
- Compilers today mostly preserve the outline of ForTran -
-
    - Lexical Analysis - recognising words/lexemes and grouping into tokens
    - Parsing - understand sentence *structure* and diagramming into trees
    - Semantic Analysis - understanding the *meaning*; reduces to equivalence checking, an undecidable problem. Involves type-checking, scope etc.
        - Limited extent performed by compilers - catches inconsistencies(types, etc.)
        - Ambiguity - variable binding(which one is being referred to?); commonly bind to the inner-most definition
        - Static analysis - program specification, concurrency safety, etc/

- Optimisation - reduce time and space complexities, power consumption, resource usage(network, disk access). It requires precise semantic modelling, various subtleties emerge
  - Can `for (unsigned i = 0; i < n + 1; i++)` → `for (unsigned i = 0; i <= n; i++)` ? No
  - Can `for (int i = 0; i < n + 1; i++)` → `for (int i = 0; i <= n; i++)` ? Yes
  - Can `(2 * i)/2` be changed to `i` ? No
  - Can `Y * 0` be changed to `0` ? No for floating point
- Code Generation - translation to low-level, machine-understandable code

---

Lexical Analysis

- The lexical analyser puts dividers between different units of the program, to identify lexemes (actual substrings from the program)
- These lexemes are then classified as various tokens (class of lexemes; a tuple of type and value). Alternately, program is broken up and each string assigned its token class, e.g. identifiers, integers, keywords, whitespaces, etc.
- Many languages like ForTran disregard whitespace, i.e. program meaning should not change on removing whitespace, e.g. `VAR 1` is same as `VA R1`
- A lexical analyser needs to
  - Recognise substrings corresponding to tokens, i.e. the lexemes
  - Recognise the token class of each of the lexemes to emit < token-class, lexeme >
- Look-ahead, in order to disambiguate, when reading the input program left-to-right, i.e. where one token ends, and the other begins.
  - `=` and `==` operators, look-ahead of one character
  - C++ - Template syntax - `Foo<Bar<Do>>` and stream syntax - `cin >> var`
  - ForTran - `DO 5 I = 1, 25` and `DO 5 I = 1.25` are loop(25 times to statement with label 5) and variable-assignment(of DO5I) respectively
  - PL/1 - `IF A THEN B` and `IF A THEN B ELSE C`, further, since no reserved words, A can be "ELSE", B can be "THEN" and so on - may even need unbounded look-ahead
- Want to minimise look-ahead, ideally have none at all

Formal Languages

- A language is a subset of all the possible strings that can be made from an alphabet, usually denoted $\sum$. Thus $L \subseteq \sum^*$
- The meaning function L maps syntax to semantics, e.g. $L(e) = M$, e could be a regular expression for the set of strings M, that it represents.
- By creating a mapping like this, we separate syntax from semantics, allowing us to change syntax without affecting semantics, as well as having redundancy - multiple syntaxes for the same semantics (e.g. decimal and roman number systems)
- The fact that L is many-to-one is the basis of optimisation, it can never be one-to-many

Regular Languages and Regular Expressions

- To keep programming languages simple, designers use *regular languages*, which are the simplest class of formal languages, and which can be analysed with a DFA
- RegExs provide a concise way of unambiguously specifying a regular language (but many ways to write the same language)
- Can be inductively defined as the set generated by operations on the languages defined by smaller regular expressions
- Base cases are - ϵ (the empty string) and single character strings
- Operations include - Union(|), Concatenations(.), Iteration/Kleene closure(*)
- Additional operators - Complement(^), Kleene Plus(+), A + ϵ (?), character ranges(a-z, A-Z, 0-9), etc.

The Algorithm

1. Construct regular expressions for each token class (keywords, numbers, identifiers, etc.) $R_1, R_2, R_3, \dots$
2. Construct a regular expression for the entire language, by taking union $R = R_1 + R_2 + R_3 + \dots$
3. For input $x_1 x_2 x_3 \dots . x_n$, for $1 \leq i \leq n$, check $x_1 x_2 \dots x_i \in L(R)$. If true then $\exists j, x_1 x_2 \dots x_i \in R_j$.
4. Remove $x_1 x_2 \dots x_i$ from the input, and repeat step 3

Points of attention

- How much input? "Maximal munch" - as much as possible
- Which token> Overlap between separate R_i and R_j leads to ambiguity - priority order, e.g. for 'if', keyword > identifier
- No match? Algorithm ends, lexing over. To avoid this, create a new token class for errors, and give it lowest priority
- Complexity? Just one pass over the input, and only few operations per character(table look-up)

Finite Automata

- Represented by $(\sum, Q, T : Q \times (\sum \bigcup \epsilon) \to Q, s, F)$ - alphabet, states, transition function, start state, final states
- Depending on T, can be deterministic or non-deterministic (DFA does not have ϵ in the transition function either)
- DFAs are deterministic as for each input, only one path, whereas NFAs accept an input if there exists some accepting path
- While both are equally powerful, non-determinism is a trade-off between space-efficiency(fewer states) and time-efficiency(more paths)
- A problem solvable in polynomial time using a DFA is P, using a NFA is NP; *P = NP?*

  Conversion of a NFA to DFA (each DFA is an NFA anyway)

  - ϵ-closure of a state = set of all states reachable through ϵ-transitions
  - Each subset of NFA states makes a DFA state, all states in an ϵ-closure reduce to one state
  - Execution of NFA leads to a set of current possible states, this is a state in the DFA
  - Exponential conversion in worst-case, NFAs are thus much smaller

  Design of a NFA for a regular language

  - ϵ - start state is accepting

- A + B - NFAs for A and B in parallel, new start and accepting states, which lead to and from the start and final states of A and B, via ϵ-transitions
- AB - A's accepting states connected to B's start state via ϵ-transitions
- A* - add new start, final and loop states, ϵ-transition from start to loop, from loop to final accepting state and A's start state, and A's final states to the loop state via ϵ-transitions

Implementation of a Finite Automaton

- DFA as a 2D Table - $T : Q \times (\sum \bigcup \{\epsilon\}) \rightarrow Q$
- Single pass over input, two look-ups per input character (one for input, another for table)
- Can share identical rows(quite common), use pointers. However, extra de-reference during look-up, again a space vs. time trade-off
- Even more space efficient, use table for NFA, $T : Q \times (\sum \bigcup \epsilon) \rightarrow \{Q_{i_1}, Q_{i_2}, \dots\}$, much smaller table, but expensive to simulate(must consider all possible paths)

Parsing

- The process of creation of a parse-tree from the sequence of tokens output by the lexer. It may be implicit, needn't be output
- For parsing, we require
  - A language describing valid strings of tokens
  - A method to distinguish valid from invalid strings of tokens
- Limitations of regular languages for parsing
  - Not powerful enough, unable to represent several common constructs like nested arithmetic constructs, nested scopes, etc.
  - These kinds of arbitrary-depth matched structures require unbounded memory to be maintained, not possible with FAs(with finite number of states)

Context-Free Languages

- A context-free grammar for a CFL can be represented as $(\sum, N, S(\in N), P)$ - the terminal symbols, non-terminal symbols, start symbol and production rules
- A production rule is a finite relation $N \rightarrow (N \bigcup \sum)^*$, and for each rule $X \rightarrow Y_1 Y_2 \dots Y_n$, $X \in N, \forall i, Y_i \in N \bigcup \sum \bigcup \{\epsilon\}$
- Terminals are called so as there are no rules to replace them; for a parser, the terminals are tokens
- A derivation of a string is a series of application of production rules on the start symbol, till no non-terminal symbols are left
  - A single step looks like $X_1 X_2 \dots X \dots X_n \rightarrow X_1 X_2 \dots Y_1 Y_2 \dots Y_n \dots X_n$
  - Replacing the left-most terminal at each step - left-most derivation. Similarly the right-most derivation
- CFL are expected to
  - Give a parse tree in addition to the language membership query answer
  - Handle errors gracefully, give feedback to the programmer

        - Be implementable by tools like bison. Many grammars can generate the same language, a CFG should have a form parsable by tools
    - Derivations can be represented as parse trees - terminals at the leaves and non-terminals in the interior nodes, with the start symbol at the root
        - In-order traversal yields the original input
        - Shows the association of operations, the priorities of operators
        - A single parse-tree may have multiple derivations (two are left and right most), differ only in the order the branches were added

    Ambiguity in CFGs

    - Multiple parse trees are possible for the same input. In this case, the grammar is said to be ambiguous. Alternately be said to be ambiguous if for a string, there exist two distinct left/right-most derivations
    - Ambiguity may leave the meaning of an expression as ill-defined, e.g. arithmetic expressions. Parser can return any valid parse tree or throw error
    - CFGs can be *disambiguated* via stratification, where the grammar is re-written to enforce precedence order of operations. Automatic disambiguation may be impossible
    - Consider the grammar for expressions (+, * and parentheses)

        > $E \to E + E | E * E | (E) | id$

        Disambiguated to

        > $E \to T | T + E$
        >
        > > $T \to id | id * T | (E) | (E) * T$

    - CFG for IF - THEN - ELSE

        > $E \to$ if E $then$ $E | if$ E $then$ E $else$ $E | \ldots$

        disambiguated to

        > $E \to MatchedIF | UnmatchedIF$
        >
        > $MatchedIF \to if$ E $then$ $MatchedIF$ $else$ $MatchedIF | \ldots$
        >
        > $UnmatchedIF \to if$ E $then$ $MatchedIF | \ if$ E $then$ $UnMatchedIF | \ if E$ $then$ $MatchedIF$ $else$ $UnmatchedIF | \ldots$

    - Ambiguity may result in a simple, more natural grammar. Thus many tools use an ambiguous grammar paired with a precedence order of operations (declaration of associativity and priority). The parser uses these to decide which move to take in case of ambiguity

---

Top-Down Parsing - Recursive Descent

- Parse tree constructed in a top-down fashion, starting from the top-level non-terminal
- Rules for the non-terminals are tried in order, as the input(tokens) is walked through left to right.
    - Can't tell with non-terminals, but as soon as terminal produced, check against input token and advance the pointer on the input token stream if there is a match

- In case of mismatch or no-match, backtrack to the last production with more options and explore(like a DFS), till all of the input is produced

Implementation

```
bool term(TOKEN tok) {  //check for a match of a given token terminal
  return *next++ == tok; //always increment the next pointer
}
bool Sn(); //return true if input matches nth production of S
bool S(); //return true if input matches any production of S
```

For the calculator grammar, the functions are like

```
bool E1() { return T(); }                              //E -> T
bool E2() { return T() && term(PLUS) && E(); }          //E -> T + E
bool E() { //for the non-terminal E
  TOKEN *save = next;
  // reset next(backtrack) and try productions - IN ORDER
  // resetting not really needed for the first production
  return (next = save, E1()) || (next = save, E2());
}

bool T1() { return term(INT); } //Check if *next points to "INT"/"Id"
bool T2() { return term(INT) && term(TIMES) && T(); }   //T -> Id * T
bool T3() { return term(LPar) && E() && term(RPar); }   //T -> (E)
bool T4() { return T3() && term(TIMES) && T(); }        //T -> (E) * T
bool T() { //for the non-terminal T
  TOKEN *save = next;
  return (next = save, T1()) || (next = save, T2()) || (next = save, T3()) || (next
= save, T4());
}
```

- Note that the first rule that returns true will cause a return due to semantics of `||`, and others wont be checked (can use this for precedence of rules for a non-terminals)
- Each `Ei, Ti` increment `next`, while `E` and `T` save the original `next` (in `save`), for backtracking
- To start parsing, initialise `next` to the first token, and simply invoke `E()`

Issues with Left Recursion

- On rules like $S \rightarrow S\ \alpha$, parser goes into infinite loop on *any* input
- Rules like $S \rightarrow S\ \alpha | \beta$ produce strings starting with β, followed by αs in a right-to-left fashion, which doesn't work with the left-to-right top-down parsing
- By stratification, the problem may be fixed. In general, the left recursive grammar rule $S \rightarrow S\ \alpha_1 | S\ \alpha_2 | S\ \alpha_3 | S\ \alpha_4 | \ldots S\ \alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m$, can be factored to $S \rightarrow \beta_1\ S' | \beta_2\ S' | \ldots | \beta_m\ S'$ and $S' \rightarrow \alpha_1\ S' | \alpha_2\ S' | \ldots | \alpha_n\ S' | \epsilon$
- Thus prior to implementing recursive descent, left-recursion must be eliminated
  - Can be done automatically (Dragon book for general algorithm)
    - Mostly done by hand, to specify the semantics associated with the productions e.g. GCC

Predictive Parsing

- Normally, the recursive descent can be exponential - due to backtracking
- Keep a look-ahead of a few tokens to "predict" which token to use, pruning the tree of possible productions. For a maximum look-ahead of k, left-to-right, the grammar is called LL(k); commonly k = 1
- For LL(1), at every step there is at most one choice for possible production
- The calculator grammar is not LL(1) as we cannot chose between the productions of T or E(two productions start with T) with one token look-ahead, regardless of non-terminals or terminals produced
- Left Factoring grammars - factor out a common prefix into a single production
  - $$X \rightarrow +\,E|\epsilon$$
    $$T \rightarrow id\,Y|(E)\,Y$$
    $$Y \rightarrow *\,T|\epsilon$$
  - Left factoring delays the decision on which production to use and in a way, artificially increases the look-ahead; we decide which production to use *after* we have seen T/id
  - By constructing the LL(1) parsing table, the parsing algorithm is simple - for the left-most non-terminal S, we look at the next input token a. Then, we simply choose the production rule at (S, a) in the table
  - Instead of using recursive functions, maintain a stack of the frontier of the parse tree. The top of the stack is the leftmost pending terminal or non-terminal. Terminals in the stack are yet to be matched in the input. Non-terminals in the stack are yet to be expanded
  - Reject on reaching the error state, accept on end of input or empty stack (PDA!)

```
init stack = <S, $$>;
init next*;
while (!stack.isEmpty()) {
  if (stack.top in Non-Terminals) {
    if (T[stack.top, *next] = Y1Y2...Yn) {
      stack.pop();
      stack.push(Y1Y2...Yn);
    }
    else error;
  }
  else { // terminal symbol at the top
    if (stack.top == *next++) stack.pop();
    else error;
  }
}
```

Constructing LL(1) parsing tables

- Fixed points - given a partial order and a set of monotonic equations, initialise all variables to the top of the order, and relax them minimally, iteratively, so that all the equations can be satisfied.
  - Final values of the variables are the fixed points
  - If the order is finite, only finite number of iterations required

- For parsing, we can represent sets of symbols as points, and edges between sets if one is a strict subset of the other
    - The "height" is the total number of tokens
    - Fixed point computation is simply a search on this graph for a vertex that satisfies all the equations
- $First(X) = \{t|X \rightarrow^* t\alpha\} \bigcup \{\epsilon|X \rightarrow^* \epsilon\}$ - if X can derive t in the first position
    - Algorithmically - fixed point computation (trivially the set of all tokens)
        - $\forall$ terminals t, $First(t) = \{t\}$
        - $\epsilon \in First(X)$ if $X \rightarrow \epsilon$ or $X \rightarrow X_1 X_2 \dots X_n$ & $\forall i, \epsilon \in First(X_i)$
        - $First(\alpha) \subseteq First(X)$ if $X \rightarrow X_1 X_2 \dots X_n \alpha$ & $\forall i, \epsilon \in First(X_i)$
- $Follow(X) = \{t|S \rightarrow^* \beta X t \delta\}$ - if t after X in *any* derivation
    - Algorithmically, again fixed point computation (trivially the set of all tokens)
        - '$'(EOF) $\in Follow(S)$
        - $\forall X \rightarrow \alpha X' \beta$, $First(\beta) - \{\epsilon\} \subseteq Follow(X')$
        - $\forall X \rightarrow \alpha X' \beta$, $Follow(X) \subseteq Follow(X')$ if $\epsilon \in First(\beta)$
- For $A \rightarrow \alpha$, T(A, t) = α if
    - $\alpha \rightarrow^* t\beta$, i.e. t $\in First(\alpha)$
    - $\alpha \rightarrow^* \epsilon$ and $S \rightarrow^* \beta A t \delta$ - when A is in stack, t in input but A can't derive t, the only option is to get rid of A by deriving ϵ from it. This only works if there is a derivation in which t follows A, i.e. $t \in Follow(A)$
- For each production $A \rightarrow \alpha$, $\forall$ terminals $t \in First(\alpha)$, T(A, t) = α
    - If $\epsilon \in First(\alpha)$ then $\forall t \in Follow(A)$, T(A, t) = α
    - If $\epsilon \in First(\alpha)$ then $\forall \$ \in Follow(A)$, T($A$, $) = α
- If the grammar is not LL(1), the parsing table will have more than one entry in at least one cell. An ambiguous, left-recursive, non left-factored grammar cannot be LL(1)

Bottom-Up Parsing - Shift-Reduce

- Parse tree constructed in a bottom-up fashion. It is more general than top-down(deterministic) but just as efficient, hence, used by most parser generation tools
- Does not have problems with left-recursive or non left-factored grammars
- Unlike recursive descent, it traces a right-most derivation - produce a right-most non-terminal at each step
    - Consequently, at a state $\alpha\beta w$, and with next step $X \rightarrow \beta$, w is a string of terminals
    - The string has two parts, left with terminals and non-terminal, right with yet unexamined terminals, e.g. $\alpha X|w$
    - Two kinds of moves - *reduce*(inverted production, e.g. for $A \rightarrow xy$, $CBxy|ijk \rightarrow CBA|ijk$) and *shift*(move | to the right by one char)
- The state $CBx|yijk$ can be modelled using a stack -
    - The left string CBx is in the stack, | signifying the top of it
    - ijk is the input not yet read from the input tape
    - The operations are modelled as -
        - *Shift* pushes a terminal onto the stack, i.e. $CBx|yijk \rightarrow CBxy|ijk$ pushes y

- *Reduce* pops of a number of symbols from the top of the stack, reduces them according to production rule, and pushes result back onto the stack, i.e. for the rule mentioned earlier, the stack goes(top-to-bottom), $CBxy \rightarrow CBA$

Shift-Reduce Conflicts

- In a given state, more than one action may lead to a valid phrase
  - If at any point, it is legal to both shift and reduce, there is a *shift-reduce* conflict.
  - If there are two reductions possible, it is a *reduce-reduce* conflict. These are always bad and usually indicate serious problems with the grammar
- To address conflicts, grammar must be re-written or a precedence order specified
- Intuitively, reduce is preferred over shift when the result can be further reduced S

Handles

- A handle is a reduction that allows further reductions back to the start state, thus reduction should happen only at handles
- During shift-reduce parsing, handles can appear only at the top of the stack, never inside
  - Trivially true for empty stack
  - Immediately after reduction(which happened at a handle), the right-most non-terminal at the top of the stack and the next handle must be to its right(as its the right-most derivation - proof by contradiction) - can reach it using shift(s)
  - Since handles always at the top, reduction always at the top of the stack

  Recognizing Handles - Viable Prefixes

- No algorithm for an arbitrary grammar; employ heuristics
- For certain grammar these heuristics always work,
  $LR(0) \subset SLR(k) \subset LALR(k) \subset LR(k) \subset Unambiguous\ CFLs \subset CFLs$, LALR(k) are general enough
- α is said to be a viable prefix for a string s if $\exists$ a valid right-most derivation $S \rightarrow^* \alpha\omega \rightarrow^* s$. In other words, $\exists$ a state $\alpha|\omega$ state during parsing - α on the stack, a portion of ω that can be looked-ahead at
  - A viable-prefix doesn't extend past the right-end of the handle. Its a *prefix of a handle*
  - As long as the prefix on the stack is viable, no parsing error has been detected
- (LR(0))Items - production with a "." somewhere in the RHS; e.g. $X \rightarrow \alpha$, can be $X \rightarrow .\ \alpha$. For ϵ productions like $X \rightarrow \alpha$, only one item, $X \rightarrow .$ possible
  - A prefix can be viable if it had complete RHS for a production
  - A stack may have many prefixes of RHS's. Let stack be $p_1, p_2, \ldots, p_{n-1}, p_n$, where p_i is the prefix of RHS of $X_i \rightarrow \alpha_i$
    - p_i should eventually reduce to X_i
    - The missing part of α_{i-1} starts with X_i, i.e. $\exists$ rules $X_{i-1} \rightarrow p_{i-1}X_i\beta$, $X_{i-2} \rightarrow p_{i-2}X_{i-1}\gamma$, etc.
    - Recursively, $p_{k+1}p_{k+2}\ldots p_{n-1}p_n$ reduces to the missing part of α_k
  - For any grammar, the set of viable prefixes is a regular language. The regular language represents the language formed by concatenating zero or more prefixes of the productions (items)

        - Conversely if a string is parsable bottom-up, then every stack state is also a viable prefix

<ADD LR and LALR parsing here>

Error Handling

- A parser must detect invalid programs and translate only the valid ones
- Error handlers should report errors accurately and clearly, recover from errors quickly and not slow down compilation of valid code
- Error handling modes for a parser -
    - Panic mode - when an error is found, discard tokens till a clear role is found, after which continue as normal
        - Can look for *synchronising tokens* like the EOS(;) in C, or the next integer(for arithmetic expressions), etc.
    - Error production - use a special error terminator, which also describes how much input to skip. Specifies common mistakes in the grammar

      > $E \rightarrow E + E | E * E | (E) | int | error\ int | (error)$

        - In the above case, tokens after the error are skipped till an int is found, or inside a parenthesised expression, discard everything till the closing bracket
        - Complicates the grammar and promotes misktakes (by making syntax less restrictive)
        - Commonly compilers give warnings about lesser mistakes
    - Error correction - find a "nearby" program, using some notion of minimal *edit distance*
        - Try token insertions/deletion upto a certain distance exhaustively
        - Hard to implement, signifcant cost to compilation time, "nearby" may not be intended program anyway
        - Commonly compilers give suggestions like in OCaml, SML, etc
        - PL/C was an error-correcting compiler, motivated by the fact that for very slow compilation times, try to find as many errors in one go

---

Semantic Analysis

- Understand the *meaning* of the program, in order to catch errors missed by parsing, which does not take into account the *context* required for many constructs

Scope Checking

- The scope of an identifier is the portion of the program in which it is accessible. The same identifier may refer to different things in different parts of the program. Different scopes for same name don't overlap
- Scope checking aims to map identifier use to its declaration
- Two options for scoping - *static* (depends only on program text/compile time), like C, or *dynamic* (depends on program behaviour/runtime), like early LISP
    - C has identifier bindings introduced by -

        - Function definitions (methods names; top-level only)
        - Function argument declarations (objects of certain types)
        - Variable declarations (objects of certain types)
        - Struct definitions (type/class names)
        - Struct field definitions (objects)
        - Typedefs (type names)
    - A dynamically-scoped variable refers to the closest enclosing binding in the execution of the program
    - In C/C++, not all identifiers follow most-closely nested rule
        - Functions definitions can't be nested in C
        - Forward declarations for functions and variables using `extern`
        - Use of member functions before declaration/definition in C++
        - Use of member field in method before definition in C++
        - Identifiers can be overridden in same scope or in different nested scope

Symbol Table

- Generally semantic analysis done as a recursive descent on AST. Need to know which identifiers are defined for the current subtree(s). Thus a data structure that tracks the current bindings of the identifiers, i.e. the *symbol table*, is maintained
- On entry to a new scope/subtree, new declarations are added to the symbol table, and after processing, on exit, they are removed, restoring the old declarations
- Usually implemented as a stack of scopes -

```
void enter_scope();    // start a new nested scope
bool check_scope(x);   // true if x defined in the top scope
symbol find_symbol(x); //search scopes stack starting from top, for x
void add_symbol(x);    // add a symbol to the top scope
void exit_scope();     // exit current scope
```

- For forward declarations, C++ style use-before-definitions, classes, etc. more than one pass over AST required (pass 1 - gather all names, pass 2 - check)

Type Checking

- A type is generally a set of values (domain), associated with a set of operations possible on those values. The goal of type checking is to ensure that operations are used with correct types, to enforce the intended interpretation of values.
- Types restrict legal programs further, by restricting operations between identifiers
    - Assembly has no type-system, so anything can be operated with anything
    - OCaml has a strict typing system with immutable variables (unlike C's mutable ones)
- Tradeoff - catching common mistakes (like multiplication of strings) vs making programs more restricted (may even lose out on optimisation).
    - Can also offload the entire responsibility onto the programmer

    - Undefined behaviour - unchecked things at compile time, that may cause the program to be undefined at run time, e.g. access out of array bounds - segfault in C
- Three kind of languages based on *when* type checking is done
    - Statically typed: Done as part of compilation, e.g. C, Java
    - Dynamically typed: Done as part of program execution, e.g. Lisp, Python, Perl
    - Untyped: No type checking. All strings in the language are valid, e.g., machine code
- Type *checking* vs. *inference* - verifying fully typed programs, with all types available vs. deducing and filling in missing type information. In C, user declares types for identifiers, and the types for expression are inferred

  Inference Rules

    - Given a set of hypotheses, draw a conclusion/inference about a type
    - $\frac{\vdash hypothesis_1, hypothesis_2, ..., hypothesis_n}{\vdash conclusion}$, if all hypotheses are provable with no assumptions, then conclusion can be proven without any assumptions
    - These rules provide templates, filling which can lead to complete typing for expressions, e.g. $\frac{\vdash e1{:}int,\ e2{:}int}{\vdash (e1 + e2){:}int}$, where $+$ can be replaced with any integer operation
        - Type checking involves building a proof tree, in a bottom-up fashion
        - This corresponds to induction on the structure of the AST, where one proof rule is used for each node
        - For a given node, the hypotheses are the proofs of each of the subexpressions, the conclusion is the type, etc. of the node/expression at the node
    - A type system is sound if $\vdash e : T$, implies, e evaluates to a value of type T. All rules should not only be sound, but also precise
    - `void` is always assumed to be a sub-type of every type

  Type Environments

    - A type environment gives types for *free* variables, used when a local, structural rule does not carry sufficient information to give type to an expression, e.g. variable ref.
    - One solution is to encode more information in the rules, maintaining a type environment - a function from object identifiers to types
    - A variable is free if it is not defined within the expression, bound if declared/defined
    - Thus the rule template now becomes $\frac{O \vdash hypothesis_1, hypothesis_2, ..., hypothesis_n}{O \vdash conclusion}$, i.e. assuming that the free variables are typed according to O, it is provable..., etc.
        - Now, the rule for variable reference goes like $\frac{\vdash O(x){=}T}{O \vdash x{:}T}$
        - Similarly for variable declaration, $\frac{O[T_0/x] \vdash e_1{:}T_1}{O \vdash \{T_0\ x; e_1\}{:}T_1}$, where $O[T_0/x](x) = T_0$ and $O[T_0/x](y) = O(y)$. This corresponds to the fact that in a new scope, new assumptions are added (about x in this case), which are removed on exit
    - The type environment is kept in the symbol table -
        - The type environment gives types to the free identifiers in the *current scope*
        - The type environment is passed down the AST from the root towards the leaves
            - To get the type of the new scope, appropriate rules are applied

        - Using the variable declaration rule, the type environment is updated for the new scope and passed to children (top-down)
    - Types are computed up the AST from the leaves towards the root (bottom-up)

Sub-Typing

- Define the $\leq$ relation on classes such that given classes $X, Y and\ Z$
    - Reflexivity - $X \leq X$
    - Anti-symmetry - $X \leq Y \iff X$ inherits from Y, and $$Y \nleq X$$
    - Transitivity - $$X \leq Y \, \and \, Y \leq Z \implies X \leq Z$$
- This gives us improved variable definition and assignment rules

  $$\frac{O \, \vdash \,e_0:T_0 \\ O[T_0/x] \,\vdash \,e_1:T_1 \\ T_0 \leq T}{O \, \vdash\, \{T\, x=e_0;\, e_1\} : T_1}$$ and $$\frac{O\, \vdash\, x:T_0 \\ O\, \vdash\, e_1:T_1\\ T_1 \leq T_0}{O \, \vdash\, x=e_1 : T_1}$$
- Interesting contrast of typing of If-Then-Else vs the ternary operator `?` -
    - The former can either make two rules depending on the condition being true or false, or make the whole thing type as void
    - For the latter, LUB of types of both branches taken and returned
- For function dispatch, maintain a mapping of function-return type-argument types, with the last argument being return type. Further, when called from an object $$e_0$$, the type of object in which the expression is, needs to be known (T)

  $$\frac{O\, \vdash\, \forall i, \, e_i:T_i\\ O\, \vdash \, C::f:(T_1', ..., T_n', T_{n +1}) \\ T_0 \lt T \\ \forall i, \,T_i \leq T_i'}{O \, \vdash\, (e_0@T).f(e_1, e_2, ..., e_n) : T_{n+1}}$$

- Implementation of type check rules is pretty straightforward, e.g. for variable def.

```
type TypeCheck(env Environment, node { T x = e0; e1 }) {
  type T0 = TypeCheck(Environment, e0);
  type T1 = TypeCheck(Environment.add(x:T0), e1);
  if (Check subtype(T0,T1))
    return T1;
}
```

Static vs. Dynamic Typing

- Type checking at compile vs run time
- Static typing proponents say:
    - Static checking catches many programming errors at compile time
    - Avoids overhead of runtime checks
    - More expressive type system constructs like templating to get around rigidity
- Dynamic typing proponents say:
    - Static type systems are restrictive. Restricts the programs that you can write (even though they may be well-typed at runtime)
        - Array de-reference quite hard to verify statically. Java decides to use runtime checks (dynamic type-checking).
    - Rapid prototyping difficult within a static type system

- Soundness theorem - for all expressions in a well-typed program, the static and dynamic types should be the same
  - In case of languages with subtyping, $$\forall{E},\,{Type_{dynamic}(E) <= Type_{static}(E)}$$
  - The static type system will not accept any incorrect program that will not pass the dynamic type check of equal power.
    - A sound static type check for array-bounds would reject incorrect programs, but it will also reject a few more that will actually pass the dynamic type check
  - Soundness of static type system - all dynamically-type-incorrect programs will be rejected. Ensured by ensuring that static type is a super-type of dynamic type.
    - A trivial static-type system that rejects all programs is sound
  - All operations that can be used on an object of type C can also be used on an object of type C' $\leq$ C, e.g. fetching attribute value, invoking a method, etc.
- Completeness - all dynamically-type-correct programs will be accepted. Not possible to ensure in general
- Dynamic subtype determined by the state of the program in execution, for 'If-Then-Else', the static type may be void, or LUB(types of all branches), but the dynamic type will be the type of one branch depending on the condition
- An example of the restrictiveness of a static type system

```
class Count {
    int i = 0; //default value = 0
    *Count inc() {
        i = i + 1;
        return *this;
  }
};
class Stock : public Count {
    string name; //name of the item
};

int main() {
    Stock a;
    Stock b = a.inc();
    ... b.name ...;
    return 0;
}
```

  - Here `a.inc()` has static type `Count`, so it fails static type check, even though it has dynamic type `Stock`, and this expression is well-typed
  - Limitation as derived classes will become unable to use `inc()` method
    - Use `dynamic_cast` - returns null if not successful, else returns a pointer of the new type at run time - effectively bypass the static type system
    - C++ allows use of templates, pass that as argument

```
template<typename T>
class Count<T> {
    int i = 0;
    T inc() {
        i <-- i + 1;
        return *static_cast<T *>(this);
    } //static_cast gets checked at compile-time!
};
class Stock : public Count<Stock>{};
```

        - This increases the expressiveness of the type system
    - Caveats -
        - Provide an "escape" mechanism in a statically-typed system, e.g. casting
            - Idea - Give control to programmer if asked for. If the programmer messes up, bad behaviour possible, and the language does not provide any guarantees
        - Dynamically typed languages retrofitted with static type-checking
            - Avoid runtime costs and to aid debugging
            - Only best-effort, no guarantees. Some dynamic checks may remain
        - Methods can be overridden, overloaded

Error Recovery

- Detecting where errors occur is easier than in parsing, simply by introducing a new type `No_type` for use with ill-typed expressions, such that `No_type` $$\leq T$$ for all types T.
    - Avoids cascading type errors due to one type-error
    - Every operation is defined for `No_Type`, with the result being `No_Type`
- The type hierarchy is not a tree anymore, it is a DAG with `No_type` at the bottom

---

Runtime Organisation

- Execution of a program is initially under the control of the operating system
    - The OS allocates space for a program
    - Code is loaded into part of the space
    - OS jumps to the entry point (e.g., "main")
- The compiler has to account for the
    - generation of the code - *both* correct and fast, doing only one is easy
    - correspondence between static (compile-time) and dynamic (run-time) structures
    - storage organisation - orchestrating the use of the data area

Data Area

- The space allocated for the program is divided into - code and data. The code area contains instructions and is usually read-only, the data is used for all the other things

The Stack - Activation Records

- Assumptions for the subsequent discussion -
  - Execution is sequential; control moves from one point in a program to another in a well-defined order (violated in the face of concurrency)
  - After procedure call, control always returns to the point immediately after the call
    - Violated in catch/throw style exceptions (an exception may escape multiple procedures before it is caught)
    - Call/cc: call with current continuation
- An activation of a procedure P is an invocation of P. Accordingly, the lifetime -
  - of an activation of P is simply all the steps needed to execute P, including nested procedure calls
  - of procedure activations is properly nested, and so can be represented as a tree
  - of a variable is the portion of execution in which it is defined
  - is a dynamic concept, unlike the scope, which is a static concept. Depends on run-time behaviour
- *Active* activations, due to proper nesting, can be tracked using a stack. Does not track all activations, only currently active ones.
  - Generally kept in the data area (in the stack), usually contiguous
  - This stack is array-like, so all activation records are kept adjacently
- Activation Record / Frame - information to, keep track of for/manage an activation
  - Need to track not only current activation info, but info of caller when needed
  - Keep arguments to the function on stack as well as the return address/control link
  - Keeping the return value on stack allows the caller to find the return value at a fixed offset from its own activation
  - Tradeoffs on which part of the activation frame should be in registers and which part in memory, as well as dividing responsibilities between caller and callee
  - The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record. Thus AR layout and the code generator must be designed together

The Heap - Objects

- Persistent objects like global variables stored specially
  - A global variable's lifetime transcends all procedure lifetimes, can't store in an AR
    - Globals are assigned a fixed address once ("statically allocated")
  - Depending on the language, there may be other statically allocated values
  - Similarly, an object may be created by a procedure that outlives it, e.g. `create_new()`, that returns an object, and this should survive de-allocation of caller's AR
  - Overall - code contains object code (read-only), static area contains data with fixed addresses (fixed sized may be writable), stack contains ARs for active procedures, which also contains locals, and heap contains all other data
  - Heap managed using `malloc` and `free` in C, can grow/shrink dynamically like stack
  - Usually stack and heap at opposite ends, and grow towards each other, if two regions touch, then program is out of memory.
    - Allows programs to use areas as they see fit - big stack or big heap
- Alignment - low-level detail, related to word size

        - Data is *word-aligned* if it begins at a word boundary
        - Most machines have some alignment restrictions - undefined behaviour, performance penalties in case of poor alignment or misaligned access
        - Padding may be needed to keep alignment, not a part of data - just unused space

Stack Machine

- Simplest model for code generation - where the only storage is stack
    - Location of the operands/result is not explicitly stated - top of stack
    - For an instruction `r = F(a1, a2, ..., an)`
        - Pop n words of the top of the stack
        - Perform the calculation `F`
        - Push the result `r` back onto the stack
- Contrast with pure register machine - provides only registers
    - More efficient due to register access instead of memory
    - Less compact instructions, e.g. `add r1,r2,r3` vs `add`. Java bytecode uses stack-based
- Compromise - n-register stack machine. In case n = 1, register is called accumulator
    - The answer is always stored in the accumulator
    - Pure stack machine - `add` needs 3 mem. ops - 2 reads and a write, 1-register stack machine needs just one memory read
    - For `r = F(a1, a2, ..., an)`, for each `ai`, evaluate it, and store result on stack. In the end, pop-off (n - 1) items of the stack, compute `r` and store result in acc. Note that `an` needn't be stored, it will be in acc. and can be used directly
    - Invariant: after evaluating an expression `e`, the accumulator holds the result and the stack is unchanged, i.e. expression evaluation preserves the stack
    - Also, the evaluation order is left to right which also determines the order of the operands on the stack - code gen. may depend on this. (In case of intermediate storage in registers, like for identifiers, may be independent)

Code Generation

- MIPS architecture
    - Prototypical Reduced Instruction Set Computer (RISC)
    - Most operations use registers for operands and results
    - Use `load` and `store` instructions to use values in memory
    - 32 general purpose registers (32 bits each), including `$sp`, `$a0`, and `$t1`
- x86 architecture
    - Complex Instruction Set Computer (CISC) with many more opcodes (~400 vs ~40)
    - Opcodes often can operate on both registers and/or memory and may not need separate load/store instructions
    - 8 general purpose registers (32 bits each) including `%esp`, `%eax`, and `%ecx`
    - Intel engineers felt that more opcodes and less registers is better. Use of on-chip real-estate for more functional units and logic (by saving space through a shorter register file and its

connections)

Invariants and Relevant Opcodes

- The accumulator is kept in MIPS register `$a0` (or x86 register `%eax`) - *holds the result*
- The stack is kept in memory - grows downwards towards lower addresses. It is *unchanged before and after* evaluation of expression/dispatch of function
- The temporary variable is kept in `$t1` for MIPS, or `%ecx` for x86
- For MIPS - the next location of the stack is kept in MIPS register `$sp`. The top of the stack is at address `$sp+4`. It is *preserved before and after* expression evaluation.
- For x86 - the top of the stack is at address `%esp`. The next location of the stack is at `%esp-4`
- MIPS opcodes -
  - `lw reg1, offset(reg2)` - reg1 = *(reg2 + offset)
  - `sw reg1, offset(reg2)` - *(reg2 + offset) = reg1
  - `addiu reg1, reg2, imm` - reg1 = reg2 + imm, without checking overflow
  - `li reg1, imm` exists, but MIPS cannot store immediate value directly to memory
- x86 opcodes -
  - `movl %reg1/(memaddr1)/$imm, %reg2/(memaddr2)` - Move 32-bit word from register `reg1` (or address `memaddr1` or the immediate value itself) into `reg2` or to memory address `memaddr2`.
  - `add %reg1/(memaddr1)/$imm, %reg2/(memaddr2)` -
    - reg2/(memaddr2) = reg1/(memaddr1)/imm + reg2/(memaddr2)
    - Overflow is always computed for both signed/unsigned arithmetic, in parallel
  - `push %reg/(memaddr)/$imm` - *(esp - 4) = reg/*memaddr/imm; esp = esp - 4
  - `pop %reg/(memaddr)/$imm` - reg/*memaddr/imm = *esp; esp = esp + 4

- Code generation, at least for expressions, can be written as a recursive descent of the AST

  Code generation function `cgen()`, looks like -

```
void cgen(i) { emit "li $a0 i" }
void cgen(e1 + e2) {
    cgen(e1);
    emit "sw $a0 0($sp)";   // push onto stack
    emit "addiu $sp $sp-4"; // increment sp
    cgen(e2);
    emit "lw $t1 r($sp)";
    emit "add $a0 $t1 $a0";
    emit "addiu $sp $sp 4"; // pop from stack
}
```

  Note : `cgen(e1+e2) = cgen(e1); "move $t1 $a0"; cgen(e2); "add $a0 $t1 $a0";` will not work, as nested calls to `cgen()` inside may clobber the register `$t1`

- Code gen for conditional branches - (new instructions `beq` and `b`)

```
void cgen(if (e1 == e2) then e2 else e4) {
```

```
    cgen(e1);
    emit "sw $a0 0($sp)";           // code gen for e1 and push result
    emit "addiu $sp $sp -4";        // store result of e1 onto stack
    cgen(e2);
    emit "lw $t1 4($sp)";           // load result of e1 into t1
    emit "addiu $sp $sp 4";
    emit "beq $a0 $t1 true_branch";  // compare e1's result with e2's (in a0)
    ...
    false_branch:
        cgen(e4);                  // cgen e4 and return
        emit  "b end_if";
    true_branch:
        emit "cgen(e3)";            // cgen e3 and return/fall through
        emit  "b end_if";
    end_if:
        return;
}
```

- Code for function calls and function definitions depends on the layout of the AR

  Activation Record design

  - The result is always in the accumulator and so no need to store the result in the AR
  - The activation record holds the actual parameters - before calling a function, arguments pushed onto the stack
  - The stack discipline guarantees that on function exit `$sp` is same as it was on entry
    - No need for a control link (which is usually needed to find another activation).
  - Return address is stored by the caller
  - A pointer to the current activation is useful, lives in the register `$fp` (frame pointer)
  - For a simple language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices (in that order)

- Jump instructions -
  - `jal label` - save address of next instruction to `$ra` and jump to label. `call` on x86 does this and saves return address on stack
  - `jr reg` - jump to address stored in register
- Code generation function `cgen()`, for function dispatch/call (caller side) -

```
void cgen(f(e1, e2, ..., en)) {
    emit "sw $fp 0($sp)";    // Build callee's AR - store own fp at top
    emit "addiu $sp $sp -4"; // i.e. push onto stack
    cgen(en);
    emit "sw $a0 0($sp)";    // save the result of args - eval each one
    emit "addiu $sp $sp - 4";// and push onto the stack, REVERSE order
    ...
    cgen(e1);
    emit "sw $a0 0($sp)";    // push the first argument
    emit "addiu $sp $sp - 4"; // sp points just below
    jal f_entry;             // jump-and-link to f_entry
    emit "addiu $sp $sp z";  // Pop-off the enire callee's frame, z = 4n + 4 + 4
```

```
    emit "lw $fp 0($sp)";    // restore the frame pointer
}
```

- The caller saves its value of the frame pointer, the actual parameters in reverse order and the return address in register `$ra`
- The size of AR so far is 4n + 8 bytes = z (+ 8 not +4 as sp points to next vacant position on stack)

- Code generation for function definition (callee side) -

```
void cgen(def f(x1, x2, ..., xn) = e) {
   f_entry:
       emit "move $fp $sp";    // fp points to base of activation of f (current
sp)
       emit "sw $ra 0($sp)";   // save return address ($ra populated by jal)
       emit "addiu $sp $sp -4";// ins_k
       cgen(e);    // code-gen, guaranteed to leave stack and sp in original state
       emit "lw $ra 4($sp)";   // since sp points one below, due to ins_k
       emit "jr $ra";          // jump to return address stored in $ra
}
```

- Note: the frame pointer points to the top, not bottom of the frame
- Either the caller or the callee, can pop-off the callee's stack frame, and restore the frame pointer
- Function calls have preserved the invariant that the stack would be exactly the same after the call, as it was at entry to the call

- Code generation for variable reference -
  - If function parameters - they are already in AR, pushed by caller and FIXED offset from `$fp` - since it points to stored return address, `$fp + 4` is the first argument, `$fp + 4` is second, and so on (reverse order useful here)
  - So for argument i, `cgen(x_i) = lw $a0 4*i($fp)`
  - Note, the location can be obtained from `$sp` as well, as long as the difference between it, and frame pointer was kept track of (extra argument to cgen() - keep track of it and call nested cgen() with appropriate changes in it)
    - Pro - one less register to keep track of (no need for $fp)
    - Cons - Harder to read/debug code, issues with variable length arrays, variadic functions, etc.
- To summarise, the activation record must be designed together with the code generator
  - Can be a *convention*(for compatibility) or can be decided at compile-time per function
  - Any decision should account for all possible callers/callees for any callee/caller.
- Production compilers -
  - Emphasize keeping values in registers (optimization), especially the current stack frame
  - Keep intermediate results in the AR (at fixed offsets for direct access)
  - Push/Pop less from the stack (more can be more expensive as no value re-use)

Temporaries

- To manage temporary values better, keep them in AR - at a fixed location within it
  - Even better is to keep them in registers
- Analyze the code to see how many temporaries are needed and allocate space for that, instead of pushing/popping from stack (thus reducing *stack traffic*)

- Many of the temporaries are only needed once and not needed after that. So some of the space can be re-used for the subsequent temporaries when evaluating
- If NT(e) are the number of temporaries required to evaluate e, then -

Expression Type	Number of Temporaries
e1 <binary operation> e2	max(NT(e1), 1 + NT(e2))
if e1 <logical operation> e2 then e3 else e4	max(NT(e1),1 + NT(e2), NT(e3), NT(e4))
id(e1, e2, ..., en)	max(NT(e1), NT(e2), ..., NT(en))
constant/identifier	0

  - For If-Then-Else - once the branch is decided, needn't hang onto e1 or e2
  - For dispatch, the space for the result of ei is in the NEW activation record, so needn't count that in the space required for the current activation record
  - For a function definition f(x1...xn) = e the AR has 2 + n + NT(e) elements - the return address, the frame pointer, n arguments and NT(e) locations for intermediate results
- Code generation must know how many temporaries are in use at each point, add argument to cgen() function - the position of the next available temporary
  - The temporary area acts like a small, fixed-size stack
  - Instead of pushing popping, all calculations done at compile time
- New cgen() becomes something like - (avoided stack manipulation instructions)

```
void cgen(e1 + e2, nt) =
    cgen(e1, nt);
    emit "sw $a0 nt($sp)";
    cgen(e2, nt + 4);
    emit "lw $t1 nt($fp)";
    emit "add $a0 $t1 $a0";
```

Object Code Generation

- If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected. The code in class A works unmodified for an object of class B.
- For A methods to work correctly in A and B objects,the attributes of A must be in the same "place" in each object

  Object Layout

  - Objects are laid out in contiguous memory
  - Each attribute stored at a fixed offset in the object. The attribute is in the same place in every object of that class. e.g. `this` points to the start offset of the object
  - When a method is invoked, the object is `this` and the fields are the attributes
  - Given a layout for class A, a layout for subclass B can be defined by *extending* the layout of A with additional slots for the additional attributes of B

        - Any method that expects to finds A's attributes at certain offsets will find them there in each of the subclasses (including B) as well
    - Given the class relationships $$A_1 \leq A_2 \leq ... \leq A_n$$, the object's attributes are laid out in the same order, starting with $$A_1$$ then $$A_2$$, ... till $$A_n$$
    - Dispatch table -
        - Every class has a fixed set of methods, including inherited ones, indexed by the dispatch table - an array of method entry points
        - A method 'f' lives at a fixed offset in the dispatch table for a class and all of its subclasses. A compiler knows all methods of a class, based on which it assigns a fixed offset for each method
            - This offset will be identical for all overriding functions of all its subclasses
            - Because methods can be overridden, the method for 'f' is not the same in every class, but is always at the same offset
        - Functions kept separately as a table, instead of living within the object itself, to save memory, as methods are less likely to be different for most objects, unlike attributes. So space is saved at the cost of one extra dereference
        - To implement a dynamic dispatch `e.f()`,
            - e is evaluated to get the class of the object, say A
            - call D[O_f], where D is A's dispatch table, and O_f is f's offset
            - In the above call, `this` is bound to A, to tell it the caller's class

Multiple Inheritance

- Let D be a class, such that $$D \leq B$$ & $$D \leq C$$
    - The object layout of D involves laying out D's header, then B's header and attributes, then C's header and attributes, and finally D's attributes
    - For each use of a D object in a context where D is expected:
        - Generate code assuming that `this` points to D's header and the object layout
    - For each use of a D object in a context where B or C is expected,
        - Generate code to convert D's `this` to B's `this` (by adding an offset to reach B or C's header), and then using the code generation for B or C respectively
    - For methods:
        - If D overrides a method of B or C, we appropriately modify B or C's dispatch table
        - While generating dispatch code in a context where D is expected, if it is an overriding method, then index into the appropriate class (e.g., B or C)
- In case $$ B \leq A$$ and $$C \leq A$$ as well (the diamond-case)
    - Can solve using one level of indirection - D's header, followed by pointers to object layouts for B and C (each of which will now contain A's header and attributes), followed by D's attributes
    - Dispatch tables will be similarly structured with two-level indirection now

---

Operational Semantics

- In the definition of a programming language -
  - The tokens → lexical analysis
  - The grammar → syntactic analysis
  - The typing rules → semantic analysis
  - The evaluation rules → code generation and optimization
- Previously the evaluation rules were specified indirectly, as the compilation of the program was to a stack machine which had certain evaluation rules
- This is an over specification. The exact specification is required as that would allow more freedom in code generation and optimization. Overspecification restricts these choices
  - Assembly-language descriptions of language implementations have irrelevant detail - use of stack machine, direction of stack growth, representation of integers, ISA of architecture, etc.
  - Thus a complete description that is high-level and not overly restrictive is required
- Another approach is reference implementations. Once again, there will be artifacts of the particular way it was implemented that wasn't meant to be part of the language
- Some ways to specify language semantics
  - Operational semantics describe program evaluation via execution rules on an *abstract machine*
  - Denotational semantic describe the program's meaning as a mathematical function
    - A mathematical function that maps input to output
    - Adds complexity through functions that isn't needed for describing an implementation
  - Axiomatic semantics describe the program's behaviour via logical formulae
    - If execution begins in state satisfying predicate X, then it ends in state satisfying Y
    - Foundation of many program verification systems. e.g., static analysis systems

Inference Rules

- Similar to typing rules, rules of inference for evaluation
  - $$\frac{Context\,\vdash hypothesis_1, hypothesis_2, ..., hypothesis_n}{Context'\,\vdash\, e:v }$$, if all hypotheses are provable within the context, then conclusion, i.e. expression e, has value $$v$$ within the context $$Context'$$
- In general, the context has the components -
  - Environment - a mapping from variables to their locations in memory (tracks current scope)
    - $$X(a_1=l_1, ..., a_n=l_n)$$ is an object with class X, attributes $$a_i$$, with locations $$l_i$$ - layout of the object is not specified, kept abstract
    - Constants do not have associated locations, only values
  - Store - a mapping from memory locations to values (objects in C++ - instances of some class)
    - $$S' = S[v'/l]$$ is a store such that $$S'(v') = l$$ and $$S'(v) = S(v), \, v \neq v'$$
    - Special value `void` - no operations can be done on it, implementations may use any value
  - Current object/ `this` / `self` - $$s_0$$
- Thus the statement $$s_0, E,S \, \vdash \, e:v, \, S'$$ means that given the current object, variable environment and store, the evaluation of e, if it terminates, yields the value $$v$$, with the store changing S'
  - Note that environment or the current object do not change, only memory does
  - "Result" of evaluation is a value and a new store, new store models the side effects

Evaluation Rules for C

<add rules here>

Intermediate Language (IR)

- Quite literally, an intermediate language between the source and the target, that provides an intermediate level of abstraction - more details than source, but fewer than target, e.g. the source language usually has no notion of registers, the IR may have them
- Experience dictates use of IR, not theory really. In fact, some compilers choose to have multiple IRs, and thus multiple lowerings (of source to IR abstraction)
    - Intuition - by making some decisions upfront (source to IR), optimization opportunity isn't (hopefully) affected much, but the reasoning about the final code generation is simplified (as the abstraction level is now closer to the target)
- IR generally ends up losing information, e.g. loop structure converted to gotos. Ideally the abstraction lowering should not loose much information, so as to not preclude much optimisation opportunity
    - Thus, the design of an IR is an art - hard to say which IR is the best (one that allows best code generation and optimisation)

Design Decisions in IR

- How high-level should the opcodes be?
    - Too low-level may preclude optimization opportunity (assembly opcodes may be higher-level than IR opcodes, e.g., vector instructions)
    - High-level IR opcodes make the IR design large and bulky and the IR starts looking almost like a CISC ISA
    - Needs to be designed for all possible ISAs, which makes the design decisions even more crucial. So the typical design choice is to support as many opcodes as necessary for all the common optimizations on all the common ISAs
- The same IR may be used for multiple high-level programming languages. e.g., LLVM may be the target for both C and Java programs. This can again increase the complexity of LLVM IR, because it must retain high-level semantics of all supported languages and making it too low-level for simplicity would preclude optimisation.
- If one can design an IR successfully, then all the implementation related to optimization needs to be done only once for all languages, and for all ISAs supported

- In the discussion that follows, it is assumed that the IR -
    - Register names or constants are operands, with no limit on temporaries (registers)
    - Opcodes & Control structures mostly correspond to assembly equivalents, but some are higher-level like `push`, `pop`, etc.
    - Three-Address Code - where all instructions are of the form `x = y op z` or `x = op y`, and `y` and `z` are registers or constants
    - Each subexpression has a "name" - allowing only one expression at a time
- IR code generation is very similar to assembly code generation, but unlimited temporaries to hold intermediate results make it simpler that use of limited registers/stack

- When it comes to optimisations, there is the question of where to do it
  - Abstract Syntax Tree
    - Pro - Machine independent
    - Con - Too high-level/abstract, need to have more details to be able to express the "kind" of machine for which the AST needs to be compiled, e.g. register, stack, etc
  - Generated Assembly code
    - Pro - Exposes optimisation opportunities
    - Cons -
      - Too low level, making optimization difficult (undo/redoing certain decisions)
      - Machine Dependent, must re-implement optimisations when re-targeting
  - IR, if designed well -
    - Pros -
      - Can be machine independent as it can represent a large family of machines
      - Can expose most optimisation opportunities
    - Con - *if designed well*, IR design critical for exposing optimisation opportunities

---

Optimisation

- Optimization seeks to improve a program's resource utilization - execution time, code size, memory/network/disk usage, power consumption, etc, but in doing so, the answer computed by the optimised program must be the same
- Largest, most complex and most compile-time-consuming part of the compiler
- "Program optimisation" is grossly misnamed, code "optimisers" make no promises of working towards the optimal program, which, for a given computation is undecidable. Code "improvers" is perhaps a better term
- Further discussion will be about optimisations on the IR with the language

> $$P \rightarrow S \space P | S$$
>
> $$ S \rightarrow $$ `id := id op id` | `id := op id` | `id := id`
>
> | `push id` | `id := pop`
>
> | `if id rel_op id goto L` | `L:` | `jump L`
>
> `op` $$\rightarrow$$ `+` | `-` | `*` | `/` | `%` | etc.
>
> `relop` $$\rightarrow$$ `<` | `>` | `==` | `!=` | etc.

- Basic block - a maximal sequence of instructions with no labels, except at the first instruction and no jumps, except in the last instruction. The idea is
  - Cannot jump into a basic block (except at beginning)
  - Cannot jump out of a basic block (except at end)
  - A basic block is a single-entry, single-exit, straight-line code segment - once the start is reached, all the instructions inside are guaranteed to execute
- Control-Flow Graph - a directed graph with basic blocks as nodes

    - Edges between nodes (blocks), if the execution can pass from the last instruction of one to the first instruction of the other
    - Edges can be due to `jump`s, `goto`s or simple fall-through
    - The body of a method can be represented as a control-flow graph. There is one initial node, the entry node, and all "return" nodes are terminal
- Typically three granularities of optimization -
    - Local optimisations, to a basic block in isolation
    - Global optimisations, to a control-flow graph, i.e. a method, in isolation
    - Inter-procedural optimisations - across method boundaries

Local Optimisations

- Optimisations on a basic block, usually the simplest. The best ones usually delete code.
- Some local optimisations -
    - Algebraic Simplification -
        - Deletion of trivial statements like `x := x + 0` or `x := x * 1`
        - Strength Reduction - Reducing the complexity of operation, useful if the "stronger" one has to be implemented in software, but the "weaker" one is supported natively by the ISA or is faster on hardware
            - `x := x ** 2` to `x := x * x`
            - `x := x * 2` to `x := x + x`
            - `x := x * 8` to `x := x << 3`
        - Less relevant on modern hardware, which deals with the special cases by itself
    - Constant Folding - operations on constants done at compile-time. If `x := y op z` and `y`, `z` are constants, then `x` needn't be computed at run-time
    - Local Dead Code Removal - elimination of unreachable basic blocks
        - Remove code not reached from the initial block - not jumped to/fallen through
            - `if (DEBUG) then { .... }`, the entire basic block may go
            - Libraries provides many methods, of which few are used, so rest of code dead
            - LHS of assignment has no further uses, then the assign. statement is dead
        - Makes the program size smaller, possibly faster - improved spatial locality and memory cache effects
- Static Single Assignment (SSA) Form - Intermediate code where assignment to each temporary occurs only once, i.e each temp. occurs only once on the LHS of assignment
    - Thus, if two assignment have the same RHS, they compute the same value
    - Common Sub-expression Elimination (CSE) - since the values of temporaries do not change, the values of the expressions involving them do not as well.
        - Three-address code means that at-most one operator can be on the RHS. For complex expressions, sub-expressions assigned to temporaries
        - Thus `y := a op b ... x := a op b` becomes `y := a op b ... x := y`
    - Copy propagation - If `x := y` occurs in basic block, replace subsequent uses of `x` with `y`. Useful for enabling constant folding (if `y` was constant) and DCE (copy statement is now dead)

- Peep-hole Optimisation - directly applied to assembly code, typically for those that got missed at IR stage (perhaps due to local minima problem)
    - At the assembly-level, greater visibility into instruction costs and instruction opcodes. e.g., hardware opcodes may be higher-level than IR opcodes (CISC machines)
    - The "peephole" is a short sequence of (usually contiguous) instructions and the optimiser replaces this sequence with another equivalent but faster one
    - Usually written as replacement rules - `i1; i2; i3; ..; in` $$\rightarrow$$ `j1; j2; ..; jm`
        - `move $a $b; move $b $a` $$\rightarrow$$ `move $a $b`, only if 2nd instruction isn't jumped to
        - `addiu $a $a i; addiu $a $a j` $$\rightarrow$$ `addiu $a $a (i+j)`
    - Many (not all) basic block optimisations can be cast as peephole optimizations
        - Algebraic Simplification - `addiu $a $b 0` $$\rightarrow$$ `move $a $b`
        - Local Dead Code Elimination - `move $a $a` $$\rightarrow$$
        - Above two together eliminate `addiu $a $a 0`
    - Peephole optimisations must be applied repeatedly for maximum effect
    - Need to ensure that the replacement rules cannot cause oscillations, each replacement rule can only "improve" the code.
- Each local optimisation can do little by itself, but optimisations interact - one optimisation may enable another. But the optimiser may also get stuck at local minimas

Register Allocation

- Intermediate code uses unlimited temporaries - simplifies code generation and optimization but complicates final translation to assembly
    - Problem - To rewrite the intermediate code to use no more temporaries than there are machine registers
    - Potential Solution - assign multiple temporaries to each register without changing the program behaviour, i.e. a many-to-one mapping
- Live-ness Analysis - A temporary is said to be live at a point after its assignment, if it is used after that point, i.e it occurs in the RHS of some expression
    - Temporaries `t1` and `t2` can share the same register if and only if at any point in the program at most one of `t1` or `t2` is live

Register Interference Graph (RIG)

- Construct the RIG as follows -
    - A node for each temporary
    - Edge between two nodes if they are live simultaneously at some program point
    - RIG extracts exactly the information needed to characterize legal register assignments - two temporaries can be allocated to the same register if there is no edge between them, and gives a global picture of the register requirements
    - Graph colouring - A colouring of a graph is an assignment of colours to nodes, such that nodes connected by an edge have different colours. A graph is *k-colourable* if it has a coloring with k colours

        - This register allocation problem is identical to the graph-colouring problem, with the colours being the registers and the graph being the RIG
    - Spilling - when the graph is not k-colourable, some temporaries will not fit in the registers available. Thus these temporaries are "spilled" into memory.
        - A node is picked (using various heuristics) as a candidate for spilling. Remove the picked node and all its incident edges from the RIG.
        - Optimistic colouring - Spill a node, k-colour the graph, try to add the node back to graph and hope the graph is still k-colourable. If it fails, no choice but to spill
        - Allocate memory location for temporary `t`, usually on the stack frame. Call it `t_a`
            - Before each operation using `t`, insert `t := load t_a`
            - After each operation updating `t`, insert `store t, t_a`
            - Rename each load/store of `t` to `t_i`, re-compute live-ness and re-try
        - After sufficient amount of spilling, the RIG should be colourable, as the graph is being made much sparser by reducing the *live range* of variables, e.g. the new variables `t_i` are live only between `load t_i` and `store t, t_i`
- Limitations of this kind of *static* register allocation -
    - All-or-nothing - does not account for variable usage of temporaries across program regions, e.g. if `t1` is used heavily in one region and sparingly in the next, spill it in the second region only
        - Region-based register allocator - partition the program into regions, assign temporaries to registers or memory within each region, and resolve discrepancies at region boundaries.
            - The overall quality of the solution depends heavily on the region partitioning
            - Heuristics include - loop region, regions based on register pressure, etc
    - Some opcodes require only register operands. Further, other opcodes may require certain specific registers. Issues arise due to the layered, pass-by-pass nature of analysis and transformation
        - Compilers add extra passes (like the "reloading" pass in GCC's register allocator) to resolve these, by adding instructions to spill/copy registers.
        - An alternative technique is to do instruction(opcode)-selection simultaneously with register-allocation. More expensive in general but can use heuristics to prune the search space while still getting most of the benefits.
            - Previous work on binary translation demonstrates an algorithm based on dynamic-programming - the performance results clearly bring out the weaknesses of the pass-by-pass register allocation approach

---

Global Optimisations - Data Flow Analysis

- Global optimisation tasks share several traits -
    - Dependency on knowing a property X at a particular point in program execution
    - Proving X at any point requires knowledge of the entire program
    - Conservative analysis - If the optimization requires X to be true, then it must be known at the program point if X is definitely true OR cannot say anything
- Global dataflow analysis is a standard technique for solving problems with the above characteristics. Every such analysis is specified by the following -

    - Domain of Values - Values that the property X can have
    - Direction - Flow of analysis - forwards(in direction of program exec.), or backwards
    - Meet Operator - Calculating the values of X at point, given several paths to/from it - must be commutative, idempotent and associative
    - Transfer Function - Relating the values of X, across a statement (transfer of info.)
    - Boundary Condition - Value of X at program entry/exit (depending on direction)
    - Initialisation - Initial value of X for every point. Because of cycles, all points must have values at all times
- Generally the algorithm of the analysis is a *fixed-point* computation
    1. Set all the values for X at the entry/exit points of program, initialise the rest
    2. For each point, repeatedly compute X using transfer function and meet operator
    3. Repeat until a fixed point has been found (no change to values of X at any point)
- Convergence of the algorithm happens if -
    - The domain of values satisfies the following -
        - It is a partial order - can be set of values, set of sets of value, etc
        - There is a $$\top$$, and a $$\bot$$ value, $$\bot < \space ... < \top$$, with a finite number of elements in between them
    - The transfer function and meet operator are such that, the $$X_{output} < X_{input}$$
    - The finiteness of the ordering ensures that for all the program points, initialised to $$\top$$, the value can change only a finite number of times, till it becomes $$\bot$$
        - The invariant is that the value at any intermediate stage of the algorithm is always greater than the best solution value
        - The relaxation is minimal - assuming the invariant is true for predecessors, it is guaranteed to be true for the successors
    - Thus, the analysis run-time is linear in program size
- Global Constant Propagation
    - To replace a use of `x` by a constant `k`, on *every* path *to* the *use of x*, the *last* assignment to `x` must be `x := k`
    - The specifics for the analysis, i.e. above property holds for variable `x`
        - Domain = $$\{\bot, c, \top\}$$, with $$\bot < c < \top$$,where these represent
            - `x` is not a constant/can't say (conservative assumption)
            - `x` = c, c being a constant (constants are incomparable)
            - The statement never executes (or not yet executed so far)
        - Direction - forward
        - Meet operator - greatest lower bound, $$glb$$
        - Transfer Function - $$f(x, s, out) = \left\{\begin{array}{ll} \top & f(x,s,in)=\top \\ c & f(x, x:=c, in) = d \space | \bot \\ \bot & s \equiv x := g(...) \\ f(x, s, in) & s \equiv y := ... \& \space x \neq y \\\end{array} \right.$$
        - Boundary Condition - $$f(x, entry, out) = \bot$$
        - Initialisation - $$f(x, ..., ...) = \top$$ for all other points
    - Now, given the global constant info. for `x`, check $$f(x, s, in) \space \forall s$$ using `x` and replace `x` in that statement, if $$f$$ its a constant

    - Since the value at any point can only reduces(according to the ordering), and thus change at most twice, so convergence is guaranteed
    - Number of steps per variable = Number of $$f(...)$$ values computed * 2 = Number of program statements * 4 (two $$f$$ values per statement, in and out)
- Global Dead Code Elimination
    - A variable `x` is live at a statement s if
        - There exists a statement s' that uses `x`.
        - There is a path from s to s', with no intervening assignment to `x`
    - Once again, the specifics of the analysis for a variable `x`
        - Domain = $$\{true, false\}$$ and $$true < false$$
        - Direction - backwards
        - Meet operator - logical OR, $$\vee$$ - live at any of the successor nodes
        - Transfer Function - $$f(x, s, in) = \left\{\begin{array}{ll} true & s \equiv ... := g(x) \\ false & s \equiv x:= e \space \& \space x \notin e \\ f(x, s, out) & x \notin s \\ \end{array} \right.$$
        - Boundary Condition - $$f(x, ..., out) = false$$
        - Initialisation - $$f(x, ..., ...) = false$$
    - For the entire set of variables, the live-ness analysis is slightly different
        - Domain = Power set of the set of all variables, the ordering is the reverse of the cardinality, i.e. $$\phi$$ is $$\top$$
        - Direction - backwards
        - Meet operator = Set Union, $$\bigcup$$
        - Transfer Function - $$X_{in}[s] = GEN(s) \cup (X_{out}[s] - KILL(s))$$, where GEN(s) are the variables used, and KILL(s) are the variables assigned, in s
        - Boundary Condition - $$X_{in}[exit] = \phi$$
        - Initialisation - $$X_{out}[.] = \phi$$
    - Once live-ness information is known, dead variable assignments can be eliminated
    - Each value can change only once (false to true), so termination is guaranteed
- Common Subexpression Elimination
    - Specification of the analysis
        - Domain = Power set of the set of *available expressions*, of the form `x = y op z`, ordering follows cardinality
        - Direction - forward
        - Meet operator - intersection, $$\bigcap$$
        - Transfer Function - $$X_{out}[s] = GEN(s) \cup (X_{in}[s] - KILL(s))$$, where GEN(s) are the subexpressions used, KILL(s) are the ones using the LHS, in s
        - Boundary Condition - $$X_{out}[entry] = \phi$$
        - Initialisation - $$X_{out}[.] =\{all \space expressions\}$$
    - SSA helps, allows maintaining a larger set of available expressions
    - Once available expression are known at each statement entry, expressions on the RHS can be replaced by a variable (LHS of available expression)
- Copy Propagation - similar to CSE, but now expression are of the form `x = y` and the substitution is of a variable, not an expression

Partial Redundancy Elimination / Lazy Code Motion

- Subsumes CSE and LICM(Loop Invariant Code Motion) among other things
- Care needed to handle *critical edges* - source has multiple successors and destination has multiple predecessors. Introduce a new basic block on it
- Pre-requisites -
  - Each statement is a basic block
  - Statements added only a beginning of basic block
  - Add a basic block on every edge with destination having multiple predecessors
- The algorithm determines where expressions are to be computed. Consists of four data-flow problems -
  - Pass 1 & 2 - Optimum computation
    - Removal of expressions not used on any execution path
    - Removal of redundancy on every execution path
  - Pass 3 - Operations executed as late as possible to minimise register usage
  - Pass 4 - Remove unnecessary copy assignments
- **Needed Expressions**
  - Domain = Power set of the set of expressions (for one expression, would have been $$\{true, false\}$$ - definitely needed and not sure)
  - Direction - backwards
  - Meet operator - intersection, $$\bigcap$$
  - Transfer Function - $$X_{in}[s] = USED(s) \cup (X_{out}[s] - KILL(s))$$, where GEN(s) are the subexpressions used, KILL(s) are the ones using the LHS, in s
  - Boundary Condition - $$X_{in}[exit] = \phi$$
  - Initialisation - $$X_{in}[.] =\{all \space expressions\}$$

  Proposal 1 - Place the expression at the frontier of "needed expressions". But it may not eliminate redundancies and may be too early (hogs a register)
- **Missing Expressions**

  An expression is missing at program point p, if its value isn't "needed" by any basic block along all paths reaching p. Will solve some redundancy problems
  - Domain = Power set of the set of expressions
  - Direction - forward
  - Transfer function - f(x) = $$X_{out}[s] = KILL(s) \cup (X_{in}[s] - Needed(s, in))$$. An expression is missing if it has been killed & not if its needed
  - Meet operator - Set Union, $$\bigcup$$
  - Boundary condition - $$X_{out}[entry] = \{all \space expressions\}$$
  - Initialisation - $$X_{out}[.] = \phi$$

  Proposal 2 - Place the expression at the earliest point it is needed and is missing. $$Earliest(s) = Needed(s, in) \cap Missing(s, in)$$. Now in the beginning, just insert a statement `t = e` $$e \in Earliest(s)$$& replace `e` by `t`. But it may still be too early.
- **Postponed Expressions**

  An expression `e` is postponed at a program point p if for all paths leading to p, the earliest placement of `e` has been seen, but not a subsequent use
  - Domain = Power set of the set of expressions

        - Direction - forward
        - Transfer function - $$X_{out}[s] = (Earliest(s) \cup X_{in}[s]) - USED(s)$$
        - Meet operator - intersection, $$\bigcap$$
        - Boundary condition - $$X_{out}[entry] = \phi$$
        - Initialisation - $$X_{out}[.] = \{all \space expressions\}$$

      Latest - frontier at the end of "postponed" propagation

        - $$Latest(s) = (Earliest(s) \cup Postponed(s, in)) \cap (USED(s) \cup E)$$
        - $$E = \bigcup_{successors(s)} (\sim Earliest(s) \and \sim Postponed(s, in)) $$
    - **Clean-up**

        - Domain = Power set of the set of expressions
        - Direction - backwards
        - Transfer function - $$X_{out}[s] = (USED(s) \cup X_{in}[s]) - latest(s)$$
        - Meet operator - union
        - Boundary Condition - $$X_{in}[exit] = \phi$$
        - Initialisation - $$X_{in}[.] =\phi$$
    - Overall

        - Backward pass to compute needed expressions
        - Forward pass to compute missing expressions
        - Earliest = intersection of needed and missing
        - Forward pass using earliest placement to compute (can be)-postponed expressions
        - Latest = frontier of (can be)-postponed expressions
        - Backward pass to eliminate unused temporary variable assignments
        - For all basic blocks b: if $$x+y \in (Latest(b) \cup Cleaned(b, out))$$:
            - Add new `t = x+y`, at beginning of b
            - Replace original `x + y` by `t` only if:
                - $$x+y \in (USED(b) \cap \sim (Latest[b] \cap \sim (Cleaned(b, out)))$$
                - It is in Latest(b) and not in Cleaned(b, out), this use of the expression does not need to be replaced by t, so let it be as it is
- Phase Ordering
    - One data-flow analysis can improve the results of another data-flow analysis -
        - Performing constant propagation/folding may replace branch predicates with constant boolean values, enabling more code to be identified as unreachable
        - Eliminating unreachable code can remove non-constant assignments to variables, thus improving the precision of constant-propagation
    - If two or more analyses are mutually beneficial, then any ordering of the analyses in which each is run only once may be sub-optimal. Simple responses include -
        - Carefully choosing an order - allow the same analysis to be performed multiple times if needed
        - Using a meta-fixed-point loop that applies all analyses in a sequence and keeps repeating it, until there is no change
    - In the presence of loops, even the above approaches may yield sub-optimal results

        - When analyzing a loop, optimistic initial assumptions must be made simultaneously for all mutually beneficial analyses to reach the best solution
        - Performing the analyses separately makes the pessimistic assumptions about the solutions of all other analyses, from which it is not possible to recover simply by iterating the separate analyses
    - The phase-ordering problem is not simply an ordering problem, but a problem that can cause optimisation to get stuck stuck in a *local* minimum regardless of orders, as compared to a super-analysis that composes the analyses together

- Optimising compilers can be though of as using a "big bag of tricks", and these "tricks" are repeatedly used until no improvement is possible
    - The optimiser can also be stopped at any point to limit compilation time
    - Certain properties on these transformations ensure convergence in this fixed point procedure. This means that the procedure is uni-directional in some sense of improvement and does not oscillate

- Often, the best/fanciest optimisations are not implemented because they may be
    - Hard to implement, too costly in compilation time
    - Have low pay-off (even though *pay-off* is vague and hard to establish, one optimisation may enable others and this is hard to predict)
