Computer Language & Compilers

Introduction
- Definition
- String, Language & Grammar
- Phases
  - Front-end & Back-end

Lexical Analysis
- Token Abstraction
- Regular Expressions
- Finite Automata
  - NFA
  - DFA

Implementation of Lexers
- From RE → NFA
- From RE → DFA Directly
- Calculate ε-Closure
- Implement NFA as Recognizer
- Implement DFA as Recognizer
- Convert NFA → DFA
- DFA Minimization

Other Issues for Lexers
- Look-ahead
- Comment Skipping
- Symbol Table

Syntax Analysis
- Parse Tree Abstraction
- Context-free Grammars
- Derivation Directions & Ambiguity

Implementation of Top-Down Parsers
- Left Recursion Elimination
- Implementing Recursive-descent Parsing
- Left Factoring: Produce $LL(1)$ Grammar
- Implementing Recursive Predictive Parsing
- Parsing Table Construction
- Implementing $LL(1)$ Parsing

Implementation of Bottom-Up Parsers
- Build $LR(0)$ Automata
- Implementing $LR(0)$ Parsing
- Implementing $SLR(1)$ Parsing
- Build $LR(1)$ Automaton
- Implementing $LR(1)$ Parsing
- Build $LALR(1)$ Automata

Other Issues for Parsers
- Conflict Resolution
- Context-sensitive v.s. Context-free
- Expressiveness Range

Error Handling
- Types of Errors
- Error Processing Rules
- Syntax Error Recovery Strategies
  - Panic Mode
  - Phrase Level
  - Error Productions
  - Global Correction

Intermediate Representations
- Definitions & Types
- Abstract Syntax Tree
- Directed Acyclic Graph
- Control Flow Graph
Introduction

Definition

Generally, a Compiler (编译器) is: "A program that takes a source-code program and translates it into an equivalent program in target language".

String, Language & Grammar

A String $s$ is a sequence of characters.

- e.g. $abc+efg-hi; 010100010$

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$st$</td>
<td>Concatenation of $s$ and $t$</td>
<td>$s\varepsilon = \varepsilon s = s$</td>
</tr>
<tr>
<td>$s^n$</td>
<td>$n$ times self-concatenations</td>
<td>$s^0 = \varepsilon$</td>
</tr>
<tr>
<td>$</td>
<td>s</td>
<td>$</td>
</tr>
</tbody>
</table>

A Language $L$ is a set of Strings over a fixed Alphabet $\Sigma$, constructed using a specific Grammar.
• e.g. \{ε, 0, 01, 011, 0111, ...\}
• Not all Strings of chars in the Alphabet is in the certain Language, only those who satisfy the Grammar rules.
  • Alphabet = \{0, 1\} and using Grammar rule RE = 01*, we can specify the above example Language
  • String 10 is then not in it

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>Empty Language</td>
<td>(\neq {\varepsilon})</td>
</tr>
<tr>
<td>(L \cup M)</td>
<td>Union of (L) and (M)</td>
<td>({s : s \in L \vee s \in M})</td>
</tr>
<tr>
<td>(L \cap M)</td>
<td>Intersection of (L) and (M)</td>
<td>({s : s \in L \land s \in M})</td>
</tr>
<tr>
<td>(LM)</td>
<td>Set of all possible concatenation results</td>
<td>({st : s \in L \land t \in M})</td>
</tr>
<tr>
<td>(L^*)</td>
<td>Zero or more self-concatenations</td>
<td>(L^+): One or more</td>
</tr>
</tbody>
</table>

A Grammar \(G\) is the description of method (rules) of how to construct a certain Language over a certain Alphabet.

• Type 0: Turing Machine \(\equiv\) Recursive Enumerable Grammar
• Type 1: Context-sensitive Grammar (CSG)
• Type 2: Context-free Grammar (CFG, 上下文无关文法), mostly recursive
• Type 3: Right-linear Grammar \(\equiv\) Regular Expressions (RE, 正则表达式), non-recursive

| Expressiveness: Type 0 > Type 1 > Type 2 > Type 3. |

Phases

A specific Phase of a compiler handles a certain task in compiling (like a module).

**Lexical Analysis** (词法分析) recognizes Words from source program.

• Works on Strings \(\rightarrow\) Produces Tokens
• Lexical Analyzer = Lexer / Scanner

**Syntax Analysis** (语法分析) recognizes abstract Sentences of Tokens.

• Works on Tokens \(\rightarrow\) Produces a Syntax-Tree
• Syntax Analyzer = Parser

**Semantic Analysis** (语义分析) checks semantic errors, and generates IR.

• Works on a Syntax-Tree \(\rightarrow\) Produces IR

**Code Generation** (代码生成) generates codes in target language.

• Works on IR \(\rightarrow\) Produces target program

Front-end & Back-end

The **Front-end** of a compiler handles analysis phases.

• Lexer + Parser + Semantic Analyzer (+ IR Generator)
• From Source Program \(\rightarrow\) Intermediate Representation

The **Back-end** of a compiler handles synthesis phases.

• (IR Optimizer +) Code Generator
• From Intermediate Representation \(\rightarrow\) Target Language

**Lexical Analysis**

**Token Abstraction**

A **Token** (词法单元) defines a category of lexemes, which play similar roles in the source program.
A Lexeme is an instance of a Token, along with its unique attributes

- e.g. 17
  - Might be an instance of an INT Token
  - Has attribute "value = 17" maybe

### Regular Expressions

A Regular Expression (RE) is a Type-3 Grammar rule.

- e.g. 01*0: \((a + b)c\)
- Has enough expressiveness to specify the composition of Tokens, thus
- We use REs for Lexical Analysis, to judge whether an input Word is a valid Token, and which kind of Token it belongs to

| Notation | Meaning | Describes Language ...
|----------|---------|--------------------------
| \(\varepsilon\) | Put an empty String here | \(L(\varepsilon) = \{ ' ' \}\) |
| \(a\) | Put a character \(a\) here | \(L(a) = \{ 'a' \}\) |
| \(r_1 + r_2\) | Either what \(r_1\) or \(r_2\) generates can appear here | \(L(r_1 + r_2) = L(r_1) \cup L(r_2)\) |
| \(r_1 r_2\) | What \(r_1\) generates concatenates with \(r_2\)'s | \(L(r_1 r_2) = L(r_1)L(r_2)\) |
| \(r^*\) | Kleen Closure of what \(r\) generates | \(L(r^*) = (L(r))^*\) |

The following are Extended Regular Expression notations (ERE, equally expressive as RE; only some shorthands).

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>([a-zA-Z])</td>
<td>Anyone in range ([a, z]) or ([A, Z])</td>
<td>(= a + \cdots + Z)</td>
</tr>
<tr>
<td>(r^+)</td>
<td>Positive Closure of what (r) generates</td>
<td>(= r(r)^*)</td>
</tr>
<tr>
<td>(r?)</td>
<td>What (r) generates appear once or not</td>
<td>(= r + \varepsilon)</td>
</tr>
<tr>
<td>(r^i)</td>
<td>What (r) generates appear (i) times</td>
<td>(= rr \cdots r, \text{ } i\text{ } \text{times})</td>
</tr>
<tr>
<td>.</td>
<td>Any single char in the whole Alphabet</td>
<td></td>
</tr>
</tbody>
</table>

A further shorthand notation is Regular Definition, which gives names to common sub-RE expressions.

- e.g. For describing integers:
  - Digit = \([0 - 9]\)
  - Integer = Digit Digit *

### Finite Automata

A Finite Automaton (p.l. -ta, 有限自动机) is a model that decides whether to accept a String as a specific kind of Token or reject it, given the RE rules.

Can be represented as:

- Transition Diagram (TD):
  - Start Arrow: an arrow marked with "start", pointing to initial state
  - State: a circle with an identifier
Notations

<table>
<thead>
<tr>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
</tr>
<tr>
<td>$S$</td>
</tr>
<tr>
<td>$s_0$</td>
</tr>
<tr>
<td>$F$</td>
</tr>
<tr>
<td>$\text{move} \ (s, c)$</td>
</tr>
<tr>
<td>$\text{eps-closure} \ (S)$</td>
</tr>
</tbody>
</table>

The $\varepsilon$-Closure of $S = S \cup \{\text{All States that can go to without consuming any input}\}$.

DFA

A Deterministic Finite Automaton (DFA) does not allow $\varepsilon$-Transitions, and for every $s \in S$, there is ONLY ONE decision for every input Symbol.

- Accepts $\sigma$ iff: there exists ONE AND ONLY ONE path from the Start State $\rightarrow$ an Accepting State that spells out $\sigma$

<table>
<thead>
<tr>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{move} \ (s, c)$</td>
</tr>
</tbody>
</table>

No $\varepsilon$-Closure concept for DFAs.

Implementation of Lexers

Each Token (described by a unique RE $\tau$) requires a unique Recognizer.

1. [WAY 1]: RE $\tau$ $\rightarrow$ NFA $\rightarrow$ Recognizer
2. [WAY 2]: RE $\tau$ $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ Recognizer
3. [WAY 3]: RE $\tau$ $\rightarrow$ DFA $\rightarrow$ Recognizer
4. [WAY 4]: RE $\tau$ $\rightarrow$ DFA $\rightarrow$ Minimized DFA $\rightarrow$ Recognizer

The Lexical Analyzer is then built from a bunch of Recognizers:
- Each Recognizer works for one Token
- Try in listed order, therefore ordering of Recognizers matters

**From RE → NFA**

Algorithm is called **Thompson's Construction**.

1. For $\varepsilon$ / each $a \in \Sigma$:

   ![Diagram](https://example.com/diagram1.png)

2. For $s + t$:

   ![Diagram](https://example.com/diagram2.png)

3. For $st$:

   ![Diagram](https://example.com/diagram3.png)

4. For $s^*$:

   ![Diagram](https://example.com/diagram4.png)

There are some requirements on such construction:

- $\mathcal{N}(s)$ and $\mathcal{N}(t)$ CANNOT have any intersections
- REMEMBER to assign unique names to all states

Properties of the resulting NFA:

- Exactly 1 Start State & 1 Accepting State
- # of States in NFA $\leq 2 \times (\# \text{ of Symbols} + \# \text{ of Operators})$ in $r$
- States do not have multiple outgoing edges with the same input symbol
- States have at most 2 outgoing $\varepsilon$ edges

**From RE → DFA Directly**

[**Step 1**]: We make Augmented RE: concatenate with symbol # (meaning “finish”).

- e.g. $(a+b)^*a#$
- Ensures at least one $\cdot$ operator in the RE

[**Step 2**]: Build syntax tree for this Augmented RE:

   ![Diagram](https://example.com/diagram5.png)

- $\varepsilon$, # and all $a \in \Sigma$ are at leaves
- All other operators are inner nodes
- Non-$\varepsilon$ leaves get its position number, increasing from left $\rightarrow$ right

[**Step 3**]: Compute $\text{nullable}()$, $\text{firstpos}()$ & $\text{lastpos}()$ for ALL nodes.

1. $\text{firstpos}(n)$: Function returning the set of positions where the first Symbol can be at, in the sub-RE rooted at $n$
2. $\text{lastpos}(n)$: Function returning the set of Positions where the last Symbol can be at, in the sub-RE rooted at $n$
3. $\text{nullable}(n)$: Function judging whether the sub-RE rooted at $n$ can generate $\varepsilon$
<table>
<thead>
<tr>
<th>n</th>
<th>nullable(n)</th>
<th>firstpos(n)</th>
<th>lastpos(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ε-leaf</td>
<td>True</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>leaf at Position i</td>
<td>False</td>
<td>{i}</td>
<td>{i}</td>
</tr>
<tr>
<td>c1 + c2</td>
<td>nullable(c1)</td>
<td></td>
<td>firstpos(c1) ∪ firstpos(c2)</td>
</tr>
<tr>
<td>c1 · c2</td>
<td>nullable(c1) &amp;&amp; nullable(c2)</td>
<td>nullable(c1) ? firstpos(c1)</td>
<td></td>
</tr>
<tr>
<td>c*</td>
<td>True</td>
<td>firstpos(c)</td>
<td>lastpos(c)</td>
</tr>
</tbody>
</table>

**[Step 4]: Compute followpos() for Leaf positions.**

`followpos(i)` : Function returning the set of positions which can follow position `i` in the generated String

Conduct a *Post-order Depth First Traversal* on the syntax tree, and do the following operations when leaving `/` nodes:

- **c1 · c2**: For all `i ∈ lastpos(c1), followpos(i) = followpos(i) ∪ firstpos(c2)``
- **c***: For all `i ∈ lastpos(c), followpos(i) = followpos(i) ∪ firstpos(c)`

**[Step 5]: Construct the DFA.**

```java
void construct() {
    S0 = firstpos(root);
    DStates = { (S0, unmarked) };
    while (DStates has an unmarked State U) {
        Mark State U;
        for (each possible input char c) {
            V = {};
            for (each position p in U whose symbol is c)
                V = Union of V and followpos(p);
            if (V is not empty) {
                if (V is not in DStates)
                    Include V in DStates, unmarked;
                Add the Transition U--c-->V;
            }
        }
    }
}
```

- A State `S` in resulting DFA is an Accepting State iff # node ∈ S
- Start State of the resulting DFA is `S0`

**Calculate ε-Closure**

Similar problem as *graph traversal.*

```java
set epsClosure(set S) {
    for (each State s in S) {
        Push s onto stack;
    }
    closure = S;
    while (stack is not empty) {
        Pop State u;
        for (each State v that u -> v is an epsilon Transition) {
            if (v is not in closure) {
                Include v in closure;
                Push v onto stack;
            }
        }
    }
    return closure;
}
```
Implement NFA as Recognizer

bool recognizer() {
  S = epsClosure(s0);
  while ((c = getchar()) != EOF)
    S = epsClosure(move(S, c));
  if (S and F has intersections)
    return ACCEPT;
  return REJECT;
}

Performance of NFA-type Recognizers: Space - $O(|\tau|)$; Time - $O(|\tau| \times |s|)$

Implement DFA as Recognizer

bool recognizer() {
  s = s_0;
  while ((c = getchar()) != EOF)
    s = move(s, c);
  if (s is in F)
    return ACCEPT;
  return REJECT;
}

Performance of DFA-type Recognizers: Space - $O(2^n)$; Time - $O(|s|)$

Convert NFA → DFA

Algorithm is called Subset Construction, since we make subset of States in original NFA into a single State in resulting DFA.

void subsetConstruction() {
  S0 = epsClosure({s0});
  DStates = {{S0, unmarked}};
  while (DStates has any unmarked State U) {
    Mark State U;
    for (each possible input char c) {
      V = epsClosure(move(U, c));
      if (V is not empty) {
        if (V is not in DStates)
          Include V in DStates, unmarked;
          Add the Transition U--c-->V;
      }
    }
  }
}

- A State $\mathcal{S}$ in resulting DFA is an Accepting State iff $\exists s \in \mathcal{S}$, $s$ is an Accepting State in original NFA
- Start State of the resulting DFA is $s_0$

DFA Minimization

Every DFA has a minimal DFA (ignoring different naming), which contains the smallest number of states.

Bipartite the original DFA states as two groups: $G_a$ - all Accepting States; $G_n$ - others

void minimize() {
  PI = {G_a, G_n};
  do {
    for (every group G in PI) {
      for (every pair of States (s, t) in G) {
        if (for every possible input char c, transition s--c--> and t--c--> go to states in the same group)
          s, t are in the same subgroup;
        else
          s, t should split into different subgroups;
    }
  }
}
A State $S$ in the minimal DFA is an Accepting State iff $\exists s \in S$, $s$ is an Accepting State in original DFA

Start State of the minimal DFA is the one containing original Starting State

Number of minimal DFAs for a Regular Language $L = | \sim_L |$, where $\sim$ means Equivalent Class

- Distinguishing Extension for $x, y$ is $z$ that EXACTLY one of $xz, yz \in L$
- $x \sim y$ (Equivalent) means no Distinguishing Extensions for $x, y$

Other Issues for Lexers

Look-ahead

For vague Languages, may need to look ahead more than one characters to determine whether to take a transition step.

- $r_1/r_2$, where $/ \Rightarrow \epsilon$ in the FA
- After determination, move lexemeBegin pointer to position of $/$ (instead of position of forward)

Comment Skipping

Comments are simply ignored. They do not interfere with the following phases.

Symbol Table

We may need a Symbol Table to hold information about Lexemes.

- Hash Table is suitable for this task
- Lexeme’s position in source file (e.g. line number) is an important information for error handling

Syntax Analysis

Parse Tree Abstraction

A Parse Tree / Syntax Tree (语法树) is a graphical representation of the structure of a program, where leaf nodes are Tokens.

- e.g.

A Parse Tree can be viewed as a Language over Tokens’ Alphabet, described by a certain CFG
- The 2nd layer of abstraction, which extracts the information of sentence structures

Context-free Grammars

A Context-free Grammar (CFG) is a Type-2 Grammar rule, which serves the construction of a Parse Tree from a stream of Tokens. We use a set of Production Rules to characterize a CFG.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A \rightarrow \alpha]</td>
<td>(A) can be replaced with (\alpha) in a step</td>
<td>Called a Production Rule</td>
</tr>
<tr>
<td>[A \rightarrow aB \mid \beta]</td>
<td>Merges two rules starting from the same Non-terminal</td>
<td></td>
</tr>
<tr>
<td>[A \Rightarrow s]</td>
<td>From Start Symbol (A), by a Production Rule, we can derive (s)</td>
<td>Called a Derivation Step</td>
</tr>
<tr>
<td>[A \Rightarrow^* s]</td>
<td>From Start Symbol (A), after Zero or more steps, can reach (s)</td>
<td>(\Rightarrow^*) means One or more</td>
</tr>
</tbody>
</table>

A **Terminal** (终结符号) is a Token; A **Non-terminal** (非终结符号) is a syntactic variable.

- The **Start Symbol** is the first one of Non-terminals; Usually represents the whole program
- A **Sentence** \(s\) is a string of Terminals such that Start Symbol \(S \Rightarrow^* s\)

A **Production Rule** (生成规则) is a law of production, from a Non-terminal to a sequence of Terminals & Non-terminals.

- e.g. \(A \rightarrow \alpha A \mid \beta\), where \(A\) is a Non-terminal and \(\alpha, \beta\) are Terminals
- May be recursive
- The procedure of applying these rules to get a sentence of Terminals is called **Sentential Form** / **Derivation**

|| Context-free Languages || > || Regular Languages ||, e.g. \(\{(i)^i : i \geq 0\}\).

**Derivation Directions & Ambiguity**

**Left-most Derivation** \((\Rightarrow_{lm})\) means to replace the leftmost Non-terminal at each step.

- If \(\beta A \gamma \Rightarrow_{lm} \beta \delta \gamma\), then NO Non-terminals in \(\beta\)
- Corresponds to **Top Down Parsing**

**Right-most Derivation** \((\Rightarrow_{rm})\) means Replace the rightmost Non-terminal at each step.

- If \(\beta A \gamma \Rightarrow_{rm} \beta \delta \gamma\), then NO Non-terminals in \(\gamma\)
- Corresponds to **Bottom Up Parsing**, in reversed manner

A **CFG** is **Ambiguous** when it produces more than one Parse Tree for the same sentence. Must remove Ambiguity for a practical CFG, by:

1. Enforce **Precedence** (优先级) and **Associativity** (结合律)
   - e.g. \(* > +\), then \(+\) gets expanded first
2. Grammar Rewritten

**Implementation of Top-Down Parsers**

**Top-Down Parsing** (Left-to-right Leftmost-derivation Parsing, **LL Parsing**) is a general, theoretical model for a parser.

![Top-Down Parsing Diagram](image)

1. **[WAY 1]**: Eliminate Left Recursion \(\rightarrow\) Recursive-descent Parsing
2. **[WAY 2]**: Eliminate Left Recursion \(\rightarrow\) Left Factoring \(\rightarrow\) Recursive Predictive Parsing
3. **[WAY 3]**: Eliminate Left Recursion \(\rightarrow\) Left Factoring \(\rightarrow\) Construct Parsing Table \(\rightarrow\) Non-recursive Predictive Parsing

**Left Recursion Elimination**

Having **Left Recursion** (左递归) means that \(\exists\) a Derivation possibility where \(A \Rightarrow^+ A\alpha\).

- Top Down Parsing CANNOT handle Left-recursive Grammars
- Can be eliminated by rewriting

For **Immediate** Left Recursions (Left Recursion that may appear in a single step), eliminate by:
For Indirect Left Recursions (Left Recursion that may appear through several Derivations), eliminate by:

```
// Non-terminals arranged in order: A1, A2, ... An. */
void eliminate() {
    for (i from 1 to n) {
        for (j from 1 to i - 1)
            Replace Aj with its products in every Production Rule Ai -> Aj ...;
        Eliminate Immediate Left Recursions Ai -> Ai ...;
    }
}
```

Implementing Recursive-descent Parsing

The most simple and general way of parsing. Needs Backtracking (回溯) every time a choice is wrong.

```
/* Example:
*  E -> T | T + E
*  T -> int | int * T | ( E )
*/
bool term(TOKEN tok)  { return *ptr++ == tok; }
bool E1()            { return T(); }
bool E2()            { return T() && term(PLUS) && E(); }
bool E() {
    TOKEN *save = ptr;
    return (ptr = save, E1()) || (ptr = save, E2());
}
bool T1()            { return term(INT); }
bool T2()            { return term(INT) && term(TIMES) && T(); }
bool T3()            { return term(OPEN) && E() && term(CLOSE); }
bool T() {
    TOKEN *save = ptr;
    return (ptr = save, T1()) || (ptr = save, T2()) || (ptr = save, T3());
}
```

Left Factoring: Produce $LL(1)$ Grammar

$LL(1)$ means Only 1 Token Look-ahead ensures which Production Rule to expand now.

To convert to a $LL(1)$ CFG, for each Non-terminal $A$:

- $A \rightarrow \alpha\beta_1 | \ldots | \alpha\beta_n | \gamma_1 | \gamma_2 | \ldots | \gamma_m$
- $A' \rightarrow \alpha A' | \gamma_1 | \gamma_2 | \ldots | \gamma_m$
- $A'' \rightarrow \beta_1 | \ldots | \beta_n$

$||LL(1)|| < ||CFG||$, so not all Grammar can be converted to $LL(1)$. Such Grammar will have an entry with multiple Production Rules to use in the Parsing Table, thus Will be inappropriate for Predictive Parsing

Implementing Recursive Predictive Parsing

No need for Backtracking since MUST be $LL(1)$ Grammar already, but still using recursions.

```
/* Example:
*  A -> a B e | c B d | C
*  B -> b B | 'epsilon'
*  C -> f
*/
void A() {
    switch (current Token) {
        case 'a': match current Token with 'a', move to next Token;
    }
```
A Parsing Table records which Production Rule to use now, when the stack top is Non-terminal \( X \), and current input Token is Terminal \( t \). With table + stack combination, we will be able to do non-recursive parsing.

**Parsing Table Construction**

### Example Parsing Table

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Input symbol</th>
<th>( \text{id} )</th>
<th>( + )</th>
<th>( * )</th>
<th>( ( )</th>
<th>( ) )</th>
<th>( S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>( E \rightarrow TE' )</td>
<td>( E \rightarrow TE' )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E'</td>
<td>( E' \rightarrow +TE )</td>
<td>( E' \rightarrow +TE )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>( T \rightarrow FT )</td>
<td>( T \rightarrow FT )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T'</td>
<td>( T' \rightarrow \epsilon )</td>
<td>( T' \rightarrow \epsilon )</td>
<td>( T' \rightarrow \epsilon )</td>
<td>( T' \rightarrow \epsilon )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>( F \rightarrow id )</td>
<td>( F \rightarrow (E) )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Step 1: Compute \( \text{FIRST}(t) \) for every Terminal and Non-terminal.

```c
void computeFirst() {
    Initialize all \( \text{FIRST}(t) \) to be an empty set;
    for (every Terminal \( t \))
        \( \text{FIRST}(t) \) is assigned to \( \{t\} \);
    do {
        for (every Production Rule \( r: X \rightarrow \ldots \) )
            if (\( r \) is \( X \rightarrow \epsilon \))
                Add 'eps' into \( \text{FIRST}(X) \);
            else
                /* Suppose \( r \) is \( X \rightarrow Y_1 Y_2 \ldots Y_k \). */
                for (i from 1 to k) {
                    \( \text{FIRST}(X) = \text{Union of \( \text{FIRST}(X) \) and \( \text{FIRST}(Y_i) \)\} \)
                    if (\( \epsilon \) is not in \( \text{FIRST}(Y_i) \))
                        break;
                }
    } while (there are updates in this iteration);
```
Checking "X \Rightarrow \epsilon?" is equivalent to Checking "\epsilon \in \text{FIRST}(X)??

\text{FIRST}(X_1, X_2, \ldots, X_k) \text{ represents } \text{FIRST()} \text{ for the stream } X_1 X_2 \ldots X_k

- e.g. If \( x_1 \) and \( x_2 \) may be \( \epsilon \), but \( x_3 \) cannot, then
- \( \text{FIRST}(x_1, x_2, \ldots, x_k) = \text{FIRST}(x_1) \cup \text{FIRST}(x_2) \cup \text{FIRST}(x_3) \)

[Step 2]: Compute \( \text{FOLLOW}() \) for every Non-terminal.

```java
void computeFollow() {
  Initialize all \text{FOLLOW()} to be an empty set;
  Add "$" into \text{FOLLOW}(Start Symbol S);
  do {
    for (every Production Rule r: X \Rightarrow Y_1 Y_2 \ldots Y_k) {
      for (i from 1 to k) {
        if (Y_i is a Non-terminal) {
          \text{FOLLOW}(Y_i) = \text{Union of } \text{FOLLOW}(Y_i) \text{ and } \text{FIRST}(Y_i+1, Y_i+2, \ldots Y_k) - \{epsilon\});
          if (i == k || epsilon is in \text{FIRST}(Y_j+1, Y_j+2, \ldots Y_k))
            \text{FOLLOW}(Y_i) = \text{Union of } \text{FOLLOW}(Y_i) \text{ and } \text{FOLLOW}(X);
        }
      }
    }
  } while (there are updates in this iteration);
}
```

[Step 3]: Build the Parsing Table.

```java
void buildParsingTable() {
  for (every Production Rule r: X \Rightarrow Y_1 Y_2 \ldots Y_k) {
    for (every possible Terminal t) {
      if (t is in \text{FIRST}(Y_1, Y_2, \ldots, Y_k))
        Add r into Table[X, t];
    }
    if (epsilon is in \text{FIRST}(Y_1, Y_2, \ldots, Y_k)) {
      for (each terminal b in \text{FOLLOW}(X)) /* "$" is also considered here. */
        Add r into Table[X, b];
    }
  }
}
```

- All empty entries are ERRORs
- If any entry contains multiple Production Rules, then the Grammar is not \( LL(1) \)

| Example of a non-\( LL(1) \) Grammar:
| \( S \rightarrow C tSE \mid a \)
| \( E \rightarrow eS \mid \epsilon \)
| \( C \rightarrow b \)

**Implementing \( LL(1) \) Parsing**

\( LL(1) \) Parsing uses a stack instead of recursions, which is more efficient, but needs a correct Parsing Table (Table-driven).

```java
bool Ll1Parser(TokenStream ts) {
  TOKEN *ip = pointer to First Token in ts;
  stack.push($);
  stack.push(Start Symbol S);
  while (true) {
    X = stack.top();
    t = *ip;
    if (X == "$") { /* Met terminator. */
      if (t == "$") return ACCEPT;
      else raise ERROR;
    } else if (X is a terminal) { /* Met a Terminal. */
```
Example procedure of $LL(1)$ Parsing:

```
if (X == t) {
    stack.pop();
    ++ip;
} else raise ERROR;
} else {
    /* Met a Non-terminal. */
    if (Table[X, t] is not empty) {
        /* Suppose Table[X, t] is X -> Y1 Y2 ... Yk. */
        stack.pop();
        for (i from k downto 1) /* Notice order. */
            stack.push(Yi);
        Output Production Rule used: X -> Y1 Y2 ... Yk;
    } else raise ERROR;
}
```

- **Example procedure of $LL(1)$ Parsing:**

<table>
<thead>
<tr>
<th>stack</th>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>$abba$</td>
<td>$S \rightarrow aBa$</td>
</tr>
<tr>
<td>$SaB$</td>
<td>$abba$</td>
<td>$S$</td>
</tr>
<tr>
<td>$SaB$</td>
<td>$baS$</td>
<td>$B \rightarrow bB$</td>
</tr>
<tr>
<td>$SaBb$</td>
<td>$baS$</td>
<td>$B \rightarrow bB$</td>
</tr>
<tr>
<td>$SaB$</td>
<td>$baS$</td>
<td>$B \rightarrow bB$</td>
</tr>
<tr>
<td>$SaBb$</td>
<td>$baS$</td>
<td>$B \rightarrow bB$</td>
</tr>
<tr>
<td>$SaB$</td>
<td>$aS$</td>
<td>$B \rightarrow e$</td>
</tr>
<tr>
<td>$Sa$</td>
<td>$aS$</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>$S$</td>
<td>accept, successful completion</td>
</tr>
</tbody>
</table>

**Implementation of Bottom-Up Parsers**

*Bottom-Up Parsing* (Left-to-right Rightmost-derivation Parsing, $LR$ Parsing) is a more practical way for implementing a parser.

- 2 important facts:
  1. Suppose $\alpha \beta \gamma$ at some step, and the next reduction will use $A \rightarrow \beta$, then $\gamma$ is a string of Terminals
  2. Suppose $\alpha A \gamma$ is reached after some step, then the next reduction will not occur at left side of $A$

- Also called *Shift-Reduce Parsing*
  - *Shift*: push next symbol onto stack top
  - *Reduce*: pop several symbols, replace with a Non-terminal, and Push back onto stack top

---

1. **[WAY 1]**: $LR(0)$ Automata $\rightarrow LR(0)$ Action & Goto Table $\rightarrow$ Parser
2. **[WAY 2]**: $LR(0)$ Automata $\rightarrow SLR(1)$ Action & Goto Table $\rightarrow$ Parser
3. **[WAY 3]**: $LR(1)$ Automata $\rightarrow LR(1)$ Action & Goto Table $\rightarrow$ Parser
4. **[WAY 4]**: $LALR(1)$ Automata $\rightarrow LALR(1)$ Action & Goto Table $\rightarrow$ Parser
**Build $LR(0)$ Automata**

The procedure of *shifting* the next Token and *reducing* at certain points is exactly like going through an Automata. Therefore we can build a $LR(0)$ Automata to do the Bottom-Up Parsing.

A **Handle** is a pair $(r, p)$, where $r$ is a Production Rule $A \rightarrow s$, and $p$ is the position of $s$ when $r$ is used in the Derivation step.

- Unambiguous Grammar has exactly one set of handles for a Right-most Derivation
- A **Viable Prefix** is a sequence that can be the stack content, which CANNOT extend past the right end of a Handle.

**Production Rule** $A \rightarrow \beta_1 \beta_2$ is **Valid** for Viable Prefix $\alpha \beta_1$ iff $S \Rightarrow^{*} \alpha A \gamma \Rightarrow \alpha \beta_1 \beta_2 \gamma$
  - If $\beta_2 = \epsilon$, should Reduce
  - If $\beta_2 \neq \epsilon$, should Shift

An $LR(0)$ **Item** $A \rightarrow \beta_1 . \beta_2$ means that:
- Production Rule $A \rightarrow \beta_1 \beta_2$ is Valid for current Viable Prefix
- We have shifted things in $\beta_1$ onto stack, but things in $\beta_2$ not met yet
- No information about next Tokens, i.e. no Look-aheads

**Step 1**: Define $\text{CLOSURE()}$ to decide States.

```c
set computeClosure(set I) {
    closure = I;
    do {
        for (every Item m in I) {
            /* Suppose m is A -> a.Xb here. */
            for (every Production Rule r: B -> c)
                Add B -> .c into closure;
        } while (there are updates in this iteration);
    } return closure;
}
```

**Step 2**: Define $\text{GOTO()}$ to decide Transitions.

```c
set computeGoto(set I, Symbol X) {
    result = {};
    for (every Item m in I) {
        /* Suppose m is A -> a.Xb here. */
        result = union of result and $\text{CLOSURE}([A -> aXb])$;
    } return result;
}
```

**Step 3**: Build $LR(0)$ Automata. Augment the Grammar by add **dummy** Production Rule $S' \rightarrow S$ first, then:

```c
void buildLR0Automata() {
    I0 = $\text{CLOSURE}([S' -> .S])$;
    DStates = {I0};
    do {
        for (each Item set I in DStates) {
            for (each Grammar Symbol X) {
                J = $\text{GOTO}(I, X)$;
                if (J is not empty) {
                    if (J is not in DStates)
                        Add J into DStates;
                    Add the Transition I--X-->J;
                }
            }
        } while (there are updates in this iteration);
    }
    /* Start State of the $LR(0)$ Automata is $I_0$ */
```

For ∀ State \( I \) containing \( S' \rightarrow S \), \( \text{GOTO}(I, \$) = \text{ACCEPT} \)

Example of a \( LR(0) \) Automata:

Conflicts may happen in Bottom-Up parsing, which indicates that current limitation on Look-aheads is too strict for this Grammar; We will need more Look-aheads to conduct Bottom-Up Parsing on such Grammar, and that may introduce more complexity to the Automata.

1. **Shift / Reduce Conflict**: both Shift and Reduce is possible for a State
2. **Reduce / Reduce Conflict**: two or more possible Reductions for a State

**Implementing \( LR(0) \) Parsing**

The idea of \( LR(0) \) Parsing is (Assume current State \( I \), next input symbol \( a \)):

- If \( X \rightarrow \alpha_1 \in I \), Reduce by \( X \rightarrow \alpha_1 \)
- If \( X \rightarrow \alpha_2, a\beta \in I \), Shift with \( a \)
- Considers no Token Look-aheads, so called \( 0 \)

A Configuration is \( (I_0X_1I_1 \ldots X_mI_m, a_1a_1+1 \ldots a_n\$) \), where:

- \( I_0X_1I_1 \ldots X_mI_m \) is current Stack content, bottom to top
- \( a_1a_1+1 \ldots a_n\$ \) is the rest of the input Token stream
- Represents:
  - A snapshot at some time in the Parsing process
  - A Right-most Derivation \( S \Rightarrow^* X_1 \ldots X_m a_1a_1+1 \ldots a_n\$ \)

We construct Action & Goto Table from \( LR(0) \) Automata, and the Parser is then straight-forward:

```c
/* Create Action Table. */
void createActionTable() {
    for (every State Ii in Automata) {
        for (every input Terminal a) {
            for (each Item r in Ii) {
                if (r is A -> B.aC)
                    Add "shift GOTO(i, a)" in Action[i, a];
                else if (r is A -> D.)
                    Add "reduce A -> D" in Action[i, a];
                else if (r is S' -> S.)
                    Add "ACCEPT" in Action[i, "$"];
            }
        }
    }
}

/* Create Goto Table. */
Goto Table is simply the GOTO function.
```
• All empty entries are ERRORs

• Conflict ⇒ Multiple Actions in 1 Action Table entry; if no Conflicts happen, then $G$ is a $LR(0)$ Grammar

• Example of an Action & Goto Table:

<table>
<thead>
<tr>
<th>LR(0) Action Table</th>
<th>Goto Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>id</td>
</tr>
<tr>
<td>0</td>
<td>\text{id}</td>
</tr>
<tr>
<td>1</td>
<td>\text{r2}</td>
</tr>
<tr>
<td>2</td>
<td>\text{r4}</td>
</tr>
<tr>
<td>3</td>
<td>\text{s5}</td>
</tr>
<tr>
<td>4</td>
<td>\text{s6}</td>
</tr>
<tr>
<td>5</td>
<td>\text{s6}</td>
</tr>
<tr>
<td>6</td>
<td>\text{s6}</td>
</tr>
<tr>
<td>7</td>
<td>\text{s6}</td>
</tr>
<tr>
<td>8</td>
<td>\text{s6}</td>
</tr>
</tbody>
</table>

○ \text{\$2} - Shift to State \text{I}_2
○ \text{\$3} - Reduce by Production Rule \#3

• Example procedure of $LR(0)$ Parsing:

```
stack | input | action | output
0 | id$id | id$id | shift 5
0idS | *id$id | reduce by F$\rightarrow$id | F$\rightarrow$id
0F3 | *id$id | reduce by T$\rightarrow$F | T$\rightarrow$F
0T2 | *id$id | shift 7
0T2*7id5 | id$id | reduce by F$\rightarrow$id | F$\rightarrow$id
0T2*7F10 | id$id | reduce by T$\rightarrow$T$^*$F | T$\rightarrow$T$^*$F
0T2 | id$id | reduce by E$\rightarrow$T | E$\rightarrow$T
0E1 | +id$ | shift 6
0E1+6id5 | id$ | reduce by F$\rightarrow$id | F$\rightarrow$id
0E1+6F3 | id$ | reduce by T$\rightarrow$F | T$\rightarrow$F
0E1+6T9 | id$ | reduce by E$\rightarrow$E$^*$T | E$\rightarrow$E$^*$T
0E1 | id$ | accept
```

Implementing $SLR(1)$ Parsing

$SLR(1)$ Parsing means "Simple" $LR(1)$, which considers 1 Token Look-ahead on Reductions (Reduce only in $FOLLOW$(current Token)). Needs a slightly different Action Table.

```cpp
/* Create Action Table. */
void createActionTable() {
    for (every State Ii in Automata) {
        for (every input Terminal a) {
            for (each Item r in Ii) {
                if (r is A$\rightarrow$B.aC)
                    Add "shift GOTO(i, a)" in Action[i, a];
                else if (r is A$\rightarrow$D. && a is in FOLLOW(A))
                    Add "reduce A$\rightarrow$D" in Action[i, a];
                else if (r is S'->S.)
                    Add "ACCEPT" in Action[i, "$$];
            }
        }
    }
}
```

• Notice that $FOLLOW$ (S') initially contains $$

• May still leave Conflicts; if no Conflicts happen, then $G$ is a $SLR(1)$ Grammar

Build $LR(1)$ Automaton

An $LR(1)$ Item $(i, a)$ is an extension of $LR(0)$ Item, where the next allowed Token $a$ is considered.

• $i$ is a $LR(0)$ Item
• $a$ is an input Terminal, allowing Reduction using $i$ when input is $a$

[Step 1]: Define $\text{CLOSURE()}$ to decide States.

```plaintext
set computeClosure(set I) {
    closure = I;
    do {
        /* Suppose $m$ is $A \rightarrow a.Bb$, $x$ here. */
        for (every Production Rule $r$: $B \rightarrow c$) {
            for (every Terminal $t$ in $\text{FIRST}(b, x)$) /* Including $\$$ symbol. */
                Add $B \rightarrow t.c$, $t$ into closure;
        }
    } while (there are updates in this iteration);
}
```

[Step 2]: Define $\text{GOTO()}$ to decide Transitions.

```plaintext
set computeGoto(set I, Symbol $X$) {
    result = {};
    for (every Item $m$ in I) {
        /* Suppose $m$ is $A \rightarrow a.Xb$, $x$ here. */
        result = union of result and $\text{CLOSURE}((A \rightarrow a.X.b, x))$;
    }
}
```

[Step 3]: Build $\text{LR(1)}$ Automaton. The dummy item here is $S' \rightarrow S.$.

• Shorthand for $r_1, a_1; r_2, a_2; \ldots ; r_n, a_n \rightarrow r_1/a_1/\ldots/a_n$
• A State will contain $A \rightarrow \alpha_1, a_1/a_2/\ldots/a_n$, where $\{a_1, a_2, \ldots, a_n\} \subseteq \text{FOLLOW}(A)$

### Implementing $\text{LR(1)}$ Parsing

By constructing $\text{LR(1)}$ Action & Goto Table, we can achieve $\text{LR(1)}$ Bottom-Up Parsing similarly.

```plaintext
/* Create Action Table. */
void createActionTable() {
    for (every State $i$ in Automata) {
        for (every input Terminal $a$) {
            for (each Item $r$ in $i$) {
                if ($r$ is $A \rightarrow B.aC, x$) /* Shift is not effected. */
                    Add "shift GOTO(i, a)" in Action[$i$, $a$];
                else if ($r$ is $A \rightarrow D.a, a$) /* Reduce only when match. */
                    Add "reduce A -> D" in Action[$i$, $a$];
                else if ($r$ is $S' \rightarrow S.$, "$\$$")
                    Add "ACCEPT" in Action[$i$, "$\$$"];
            }
        }
    }
}
```

• May still leave Conflicts; if no Conflicts happen, then $G$ is a $\text{LR(1)}$ Grammar

### Build $\text{LALR(1)}$ Automata

A Core is the set of all $\text{LR(0)}$ items in a $\text{LR(1)}$ State, ignoring the following Terminal symbol.

$\text{LALR(1)}$ merges all the $\text{LR(1)}$ states with the same Core.

• Is a Trade-off between Grammar range ($\text{LR(1)}$) v.s. Efficiency ($\text{SLR(1)}$)
  - Number of States in $\text{LALR(1)}$ Automata = Number of States in $\text{SLR(1)}$ Automata
  - Will only introduce Reduce / Reduce Conflicts into original $\text{LR(1)}$ Parser; if no Conflicts happen, then $G$ is a $\text{LALR(1)}$ Grammar
• Used in "YACC/Bison"

### Other Issues for Parsers
Conflict Resolution

Conflicts cannot be 100% removed in LR Parsing; Also, Ambiguous Grammars are sometimes more human-readable. The possible solutions are:

1. Use context informations from Symbol Table
2. Always in favor of Shift
3. Use Precedence & Associativity, e.g.
   - $E + E$, met $+$, do Reduce since $+$ is left-associative
   - $E + E$, met $*$, do Shift since $*$ has higher precedence
   - $E * E$, met $+$, do Reduce since $*$ has higher precedence
   - $E * E$, met $*$, do Reduce since $*$ is left-associative
4. Grammar Rewriting

Context-sensitive v.s. Context-free

NOT Context-free Language $= \text{CANNOT write a CFG for this Language.}$

- e.g. $\{\omega w : \omega \in L((a + b)^*)\}$

CFG is not closed under all Language operations. Closed under $L_1 \cup L_2$, $L_1 L_2$, but NOT closed under $L_1 \cap L_2$.

Expressiveness Range

The expressiveness range of CFGs follow the relation:

<table>
<thead>
<tr>
<th>Context-free grammars</th>
<th>Unambiguous CFGs</th>
<th>Operator precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>LR(1)</td>
<td>LL</td>
</tr>
<tr>
<td>LR(1)</td>
<td>LALR(1)</td>
<td>LL(1)</td>
</tr>
<tr>
<td>LL</td>
<td>SLR(1)</td>
<td>LR(0)</td>
</tr>
</tbody>
</table>

Error Handling

Types of Errors

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Example</th>
<th>Detector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical</td>
<td>$x # y = 1$</td>
<td>Lexer</td>
</tr>
<tr>
<td>Syntax</td>
<td>$x = 1 \ y = 2$</td>
<td>Parser</td>
</tr>
<tr>
<td>Semantic</td>
<td>int $x; \ y = x(1)$</td>
<td>Type Checker</td>
</tr>
<tr>
<td>Correctness</td>
<td>Can compile, but wrong output</td>
<td>User / Static Analysis / Model Checker / ...</td>
</tr>
</tbody>
</table>

Error Processing Rules

1. Detect Errors
2. Find the positions where they occur
3. Accurately present them to users
4. Recover / Pass over to continue to find later errors
5. Do NOT impact compilation of correct part of the program

Syntax Error Recovery Strategies

Panic Mode

Discard wrong input Tokens until an expected Token is met.

- e.g. $(1 + + 2) * 3 \Rightarrow \text{skip} +$
• For LL Parsing:
  - Synchronizing Token: Terminals in FOLLOW(stack_top)
  - Skipping input symbols until a Synchronizing Token is found

• For LR Parsing:
  1. Skipping input symbols
  2. Popping stack items

**Phrase Level**

Local (Intra-sentence) correction on the input.

- e.g. \( x = 1 \ y = 2 \Rightarrow \text{insert } ; \)

• For LL Parsing:
  - Each empty entry in Parsing Table is a pointer to *specific* error routine
  - Can design whether to insert / delete / . . . symbols

• For LR Parsing:
  - Each empty entry in Action Table is a pointer to *specific* error routine

**Error Productions**

Add Production Rules specially for *typical* Errors.

- e.g. Add \( E_1 \rightarrow ID := Expr \) in Grammar for \( C \)

• Used in "GCC"

**Global Correction**

Globally analyze and find the Errors. Too ideal and hard to design.

**Intermediate Representations**

**Definitions & Types**

An Intermediate Representation (IR) is an intermediate (neither source nor target) form of a program. There are various types of IRs:

- Structural
  - Abstract Syntax Trees (AST)
  - Directed Acyclic Graphs (DAG)
  - Control Flow Graphs (CFG)
  - Data Dependence Graphs (DDG)

- Linear
  - Static Single Assignment Form (SSA)
  - 3-address Code
  - Stack Code

There will be hybrid combinations, and which to choose strongly depends on the design goals of the compiler system.

**Abstract Syntax Tree**

AST is a simplified Parse Tree.

*Example:*

```
  E  
  |  
  E  
  |  
  E 
  / 
 id:x 
/    
num:2 
 
  (id:y) 
  /    
(num:2) 
  /    
(id:y) 
```

• Advantages
  - Close to source code
  - Suitable for source-source translation
Disadvantages

- Traversal & Transformations are expensive
- Pointer-intensive
- Memory-allocation-intensive

**Directed Acyclic Graph**

DAG is an optimized AST, with identical nodes *shared*.

*Example:*

```
(id: x)
+ ->
(id: y)
+ -> (num: 2)
(id: x)
```

**Advantages**

- Explicit sharing
- Exposes redundancy, more efficient

**Disadvantage**

- Difficult to transform
- Analysis usage > Practical usage

**Control Flow Graph**

CFG is a flow chart of program execution. It is a conservative approximation of the Control Flow, because only one branch will be actually executed.

A **Basic Block** is a consecutive sequence of Statements $S_1, \ldots, S_n$, where flow must enter this block only at $S_1$, AND if $S_1$ is executed, then $S_2, \ldots, S_n$ are executed strictly in that order, unless one Statement causes halting.

- The **Leader** is the first Statement of a Basic Block
- A **Maximal Basic Block** is a maximal-length Basic Block

**Nodes** of a CFG are Maximal Basic Blocks, and **Edges** of a CFG represent control flows

- $\exists$ edge $b_1 \rightarrow b_2$ iff control may transfer from the last Statement of $b_1$ to the first Statement of $b_2$

*Example:*

```
if x = y then
  S1
else
  S2
end
```

**Single Static Assignment**

SSA means every variable will only be assigned value ONCE (therefore *single*). Useful for various kinds of optimizations.

*Example:*

```
X := 3;
X := X + 1;
X := 7;
X := X*2;
```

A $\phi$-function generates an extra assignment to "choose" from Branches or Loops. If Basic Block $B$ has Predecessors $P_1, \ldots, P_n$, then $X = \phi(v_1, \ldots, v_n)$ assigns $X = v_j$ if control enters $B$ from $P_j$. 
1. 2-way Branch:
   ```
   if (...) X = 5;
   else X = 3;
   Y = X;
   ```

2. While Loop:
   ```
   j = 1;
   s: while (j < x) {
     if (j >= X) goto E;
     j = j+1;
   }
   goto s;
   ```

- $\phi$ is not an executable operation
- Number of $\phi$ arguments = Number of incoming edges

**Where to place a $\phi$?**

- If Basic Block $B$ contains an assignment to variable $X$, then a $\phi$ MUST be inserted before each Basic Block $Z$ that:
  1. $\exists$ a non-empty path $B \rightarrow^* Z$
  2. $\exists$ a path from ENTRY to $Z$ which does not go through $B$
  3. $Z$ is the FIRST node that satisfies i. and ii.

**Stack Machine Code**

Stack Code is used for stack architectures / Bytecodes.

Example:
```
x = 2 * y - 2 * z;
push x
push 2
push y
multiply
push 2
push z
multiply
add
subtract
``` 

- Advantages
  - Compact Form
  - Names are implicit, therefore no need for temporary variables
  - Simple to generate and execute
- Disadvantages
  - Does not match current architectures
  - Difficult to reorder
  - Cannot reuse expression values, slow & hard to optimize

**3-address Code**

3-address Code takes 1 Operator + at most 3 Operands for each Statement (therefore 3-address).

Example:

<table>
<thead>
<tr>
<th>assignments</th>
<th>x = y op z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x = op y</td>
</tr>
<tr>
<td></td>
<td>x = y[i]</td>
</tr>
<tr>
<td></td>
<td>x = y</td>
</tr>
<tr>
<td>branches</td>
<td>goto L</td>
</tr>
<tr>
<td>conditional branches</td>
<td>if x relop y goto L</td>
</tr>
<tr>
<td>procedure calls</td>
<td>param x</td>
</tr>
<tr>
<td></td>
<td>param y</td>
</tr>
<tr>
<td></td>
<td>call p</td>
</tr>
<tr>
<td>address and pointer assignments</td>
<td>x = &amp;y</td>
</tr>
<tr>
<td></td>
<td>y = z</td>
</tr>
</tbody>
</table>
- **Quadruples**: Uses explicit names to store results. Easy to reorder, but needs more fields.

- **Triples**: Table indices used as implicit names. Harder to reorder, but needs less fields.

**IR Choosing Strategies**

1. **High-level Models**
   - Retain high-level data types (e.g. Classes)
   - Retain high-level control infos
   - Operate directly on program variables (NOT registers)

2. **Mid-level Models**
   - Retain part of high-level data types (e.g. Arrays)
   - Linear Code + CFG
   - Uses virtual registers

3. **Low-level Models**
   - Linear memory model, no high-level data types
   - Explicit addressing
   - Exposes physical registers

**Semantic Analysis**

**Attributes**

To add semantic information beyond the Sentence structure, we need to attach **Attributes** to Parse Tree nodes. Attributes can reveal additional informations about that node’s type (most important semantic info), value (not always needed), an so on.

**Synthesized Attributes** like $A. \text{syn}$ are synthesized using $s’$ (children’s) Attributes

- e.g. $A \rightarrow \alpha_1 + \alpha_2, A.\text{val} = \alpha_1.\text{val} + \alpha_2.\text{val}$
- $A$’s Attribute $\text{val}$ is synthesized from children’s $\text{vals}$

**Inherited Attributes** like $\alpha_1. \text{in}$ are inherited (passed down) from $A$’s (parent’s) Attributes

- e.g. $L \rightarrow L_1, \text{id}, L_1.\text{type} = L.\text{type}$
- $L_1$’s Attribute $\text{type}$ is inherited from $L$’s $\text{type}$

**Syntax-Directed Definitions**

In **Syntax-Directed** (语法制导) Definitions, a Production Rule $A \rightarrow \alpha_1 \alpha_2$ is related to a set of Semantic Rules, which give relations of Attributes of nodes on that Production Rule.

- e.g. $A.\text{syn} = f(\alpha_1. x, \alpha_2. x); \alpha_1. \text{in} = g(A. x)$

They are just related informations, but do not carry any hints for evaluation.

If there is a Semantic Rule $b = f(c_1, c_2, \ldots, c_n)$, then $b$ is dependent on $c_1, c_2, \ldots, c_n$.

- This Semantic Rule must be evaluated AFTER Rules for $c_1, c_2, \ldots, c_n$
- Dependency can be represented by a directed **Dependency Graph**

1. Mark the AST with Semantic Rules
2. Each Semantic Rule gets an id
3. Draw dependency relations between Rules
4. Verify that it is Acyclic

- e.g.

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T L</td>
<td>L.in = T.type</td>
</tr>
<tr>
<td>T → int</td>
<td>T.type = integer</td>
</tr>
<tr>
<td>T → real</td>
<td>T.type = real</td>
</tr>
<tr>
<td>L → L₁, id</td>
<td>L₁.in = L.in, addtype(id.entry,L.in)</td>
</tr>
<tr>
<td>L → id</td>
<td>addtype(id.entry,L.in)</td>
</tr>
</tbody>
</table>

**Evaluation**

**Input:** real id₁, id₂, id₃

**S-Attributed** Definitions only use Synthesized Attributes.

**L-Attributed** Definitions require that in each Production Rule \( A \to \alpha₁ \alpha₂ \ldots \) with Semantic Rule \( b \to f(c₁, c₂, \ldots, cₙ) \):

- \( b \) is a Synthesized Attribute of \( A \), OR
- \( b \) is an Inherited Attribute of \( \alpha_j \), which depends no more than Attributes of \( A \), \( \alpha₁, \ldots, \alpha_{j-1} \)

**Evaluation of Semantic Rules**

**Parse-tree Method** (General):

1. Build the AST by Parsing
2. Build the Dependency Graph from AST, verify it is a DAG
3. Obtain a workable evaluation order by Topological Sort
4. Conduct the Rules in that order

**Predetermined Evaluation** (Bottom-Up Evaluation):

- Require strictly restricted S-Attributed Definitions, but can be done along with Parsing
- Uses an additional Value Stack
  - Push in its `real` when shifting by a valued Token (e.g. `int`, 3)
  - Push in a `=` (占位符) when shifting by an unvalued Token (e.g. `+`)
  - Pop out values and Push in the result when reducing

**Translation Schemes** (i.e. Syntax-directed Translation):

- Less restricted, using L-Attributed Definitions, while also can be done along with Parsing
- Every time the Parser meets a Semantic Action, evaluate it

**Syntax-directed Translation**

In **Syntax-directed Translation** (语法制导翻译), Semantic Rules are enclosed between \( \{} \) and inserted within Production Rules.

- Semantic Rules enclosed between \( \{} \) are called **Semantic Actions**
- Position of a Semantic Action indicates when it is evaluated

**Translation Schemes Design**

With the property of L-Attributed Definitions, we can organize the positions of Semantic Actions as:

- For a Synthesized Attribute, put the action in at the end
- For an Inherited Attribute of \( \alpha_j \), put the action just before \( \alpha_j \)
- e.g.

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>D → T { L.in = T.type } L</td>
<td></td>
</tr>
<tr>
<td>T → int { T.type = integer }</td>
<td></td>
</tr>
<tr>
<td>T → real { T.type = real }</td>
<td></td>
</tr>
<tr>
<td>L → id { addtype(id.entry,L.in), L₁.in = L.in } L₁</td>
<td></td>
</tr>
<tr>
<td>L → ε</td>
<td></td>
</tr>
</tbody>
</table>

**Left Recursion Elimination**
When there are Left Recursions in the decorated Production Rules, and we want to conduct Top-Down Parsing, we will need to correctly eliminate them by:

- \[ A \to A_1 Y \{ A.a = g(A_1.a, Y.y) \} \]
- \[ A \to X \{ A.a = f(X.x) \} \]
- \[ A \to X \{ A.a = f(X.x) \} \]
- \[ A \to X \{ A'.in = f(X.x) \} A' \{ A.a = A'.syn \} \]
- \[ A \to Y \{ A'.in = g(A'.in, Y.y) \} A' \{ A'.syn = A1'.syn \} \]
- \[ A \to \varepsilon \{ A'.syn = A'.in \} \]

**Scoping**

Scoping refers to the issue of matching identifier Declarations with its Uses. The Scope of an identifier is the portion of a program where it is accessible.

- Same identifier may refer to different things in different portions
- Different scopes for same identifier name DO NOT overlap
- Usually, search for local definitions first, and if not found, goto its parent Scope

**Static Scoping v.s. Dynamic Scoping**

On **Static Scoping**, depends only on text, not runtime behavior.

- May obey Closest Enclosing Definition
  - Can be nested
  - Refer to closest parent definition
- May obey Globally Visible Definition
  - CANNOT be nested
  - Can be used before defined

On **Dynamic Scoping**, may depend on the closest binding during execution.

**Symbol Tables**

We have a separate **Symbol Table** for each Scope, where:

- Child Scope points to its Parent Scope
- May need multiple passes to generate (to serve **Globally Visible Definitions**)
- e.g.

```
Int x,y;
Procedure P:
  Bool x, a;

Procedure Q:
  Real x, y, z;
  begin
  end
begin
end
```

**Type Systems**

The **Type System** of a Language specifies:

- **Type Checking**: which operations are valid for which types
- **Type Inference**: infer the implicit type informations, i.e. decorate the Parse Tree with full type informations

Type System are based on Rules of Inference, and may not be perfectly correct. We call it:

- **Sound**: means no False Positive
- **Complete**: means no False Negative

**Language Typing Categories**

Different Languages have different strategy for Typing:
• In Statically Typed Languages, type checking is done as part of compilation (e.g. C, Java, Rust, COOL).
• In Dynamically Typed Languages, type checking is done as part of program execution (e.g. Python, Scheme).
• In Untyped Languages, there lies NO types (e.g. Machine Codes).

Rules of Inference

We Use Rules of Inference like

\[ |-\text{Hypothesis}_1, \ldots, |-\text{Hypothesis}_n \]
\[ \Rightarrow \text{Conclusion} \]

when each Hypothesis \( H \) and the Conclusion are in the form \( \text{Context} \vdash \text{expr} : T \).

To achieve effective inferences for Languages like COOL, we must introduce the following Contexts:

• **Type Environment** \( O \): a function giving types for Free Variables
  - e.g. \( O(x) = \text{Int} \)
  - Variable \( x \) is Free if it is not defined within current expression
  - \( O[T/x] \) means to update \( O \) by adding information \( O(x) = T \)
    - Needed for \( \text{let} / \text{case} \) Expressions, since they introduce new variable names in a new sub-scope

• **Method Environment** \( M \): needed for method dispatches
  - e.g. \( M(C, f) = (T_1, \ldots, T_n, T) \)
  - Means that in class \( C \), method \( f \) takes parameters of type \( T_1, \ldots, T_n \), and returns type \( T \)

• **Self-class Environment** \( C \): current \( \text{SELF\_TYPE} \) class, needed for handling \( \text{SELF\_TYPES} \)
  - Means we are inside Class \( C \) now
  - Properties:
    - \( \text{SELF\_TYPE}_C \leq C \)
    - \( \text{SELF\_TYPE}_C \leq T \) if \( C \leq T \)
    - \( \text{lub}(\text{SELF\_TYPE}_C, T) = \text{lub}(C, T) \)

Several additional rules are introduced to serve Inheritance:

• **Subtyping** \( X \leq Y \) means Type \( X \) can be used when Type \( Y \) is expected
  - Properties:
    - \( X \leq X \)
    - \( X \leq Y \) if \( X \) inherits \( Y \)
    - \( X \leq Z \) if \( X \leq Y \) AND \( Y \leq Z \)
  - Soundness Theorem: \( \forall E, \text{dynamicType}(E) \leq \text{staticType}(E) \), where:
    - Dynamic Type is the run-time evaluated type of an Expression
    - Static Type captures all possible Dynamic Types

• Least Upper Bounds: \( \text{lub}(T_1, \ldots, T_n) \) means the smallest parent class of all \( T_1, \ldots, T_n \)
  - Needed for \( \text{case} \) branches

Static Type Checking Strategy

COOL Type Checking can be done along with a tree traversal over AST (suppose we already have the global inheritance informations).

1. Type Environments \( O, M, C \) are passed down the AST
2. Type Derivations are conducted bottom up the AST towards root
   - e.g.

   \[ \begin{align*}
   |- \text{false} : \text{Bool} \quad & \quad |- 2 : \text{Int} \quad & \quad |- 3 : \text{Int} \\
   |- \text{not false} : \text{Bool} \quad & \quad |- 1 : \text{Int} \quad & \quad |- 2 + 3 : \text{Int} \\
   |- \text{while not false loop } 1 + 2 * 3 \text{ pool:Objec}t
   \end{align*} \]

   For detailed COOL Typing Rules, refer to COOLAid Manual, section 12.

Code Generation
Operational Semantics

Formal Semantics are unambiguous abstractions of how the program is executed on a machine. They guide the implementation of Code Generators.

One kind of Formal Semantics is Operational Semantics, where we use Operational Rules to demonstrate the effect of every possible operation. Similar to Type Systems, these rules are in the form of Rules of Inference, but different Contexts are needed, and the thing we infer is Value \( v \) instead of Type \( T \), along with a new Store.

- **Environment** \( E : E(x) = l_x \) tells the address (location) in memory where \( x \)'s value is stored
  - e.g. \( E = [x : l_x, y : l_y] \)
  - Will never change after an operation
- **Store** \( S : S(l_x) = v \) tells the value stored in location \( l_x \)
  - e.g. \( S = [l_2 : 2, l_y : 0] \)
  - \( S[v/l_x] \) means to update \( S \) by adding information \( S(l_x) = v \)
    - Needed for \texttt{let} / \texttt{case} expressions, since they introduce new variables in new sub-scopes
    - A Rule may have side effects: change the Store
- **Self-object** : current \texttt{self} object, needed for inferring \texttt{self}
  - Will never change after an operation

Specially for COOL, where everything are Objects, we denote a value as \( v = T(a_1 = l_1, \ldots, a_n = l_n) \).

- \( T \) is the Dynamic Type of value \( v \)
- \( a_i \) is the \( i \)th Attribute, where the location of \( a_i \)'s value is \( l_i \)

Special notations for basic classes:

1. \texttt{Int(5)}: integer value 5
2. \texttt{Bool(true)}: boolean value true
3. \texttt{String(\text{"Cool"})}: string \text{"cool"} with length 4
4. \texttt{void}: special instance of all types, only effective for \texttt{isvoid}

Several additional rules are introduced for new objects and method dispatches:

- \( l_{\text{new}} = \texttt{newloc}(S) \) means allocate a new, free location \( l_{\text{new}} \) in memory
  - Needed for \texttt{let} / \texttt{case} / \texttt{new} expressions, since they ask for new objects
  - Hides some details like the size and strategy of allocation
- \( D_T \) means the default value object of Type \( T \)
- \texttt{class}(\( T \)) = \((a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n) \) illustrates the composition of Type \( T \)
  - Needed for new expressions
- \texttt{impl}(\( T, f \)) = \((e_1, \ldots, e_n, e_{\text{body}}) \) illustrates the composition of Method \( T.f \)
  - Needed for method dispatches

For detailed COOL Operational Semantics, refer to COOLAid Manual, section 13.

There are other kinds of more theoretical and abstract Formal Semantics, e.g.

- Denotational Semantics
- Axiomatic Semantics

Runtime System

The Runtime System (Environment) defines the way of managing run-time resources. It depends largely on the machine architecture and OS.

- **Memory Layout and Usage:**
  - Allocation and Layout of objects
  - Function call strategies
  - Garbage collection or not, and how
- **Convention of using Registers**
- **Runtime Error handling API**
To generate workable code, we **MUST** obey *uniform* routines with the Runtime System definitions when implementing the Code Generator. Thus, Code Generator design **MUST** consider the run-time requirements of the target machine and OS.

For the detailed COOL Runtime System Conventions, refer to COOL Runtime System, section 2-5.

- In object layouts, subclasses arrange its attributes from the oldest ancestor's (i.e. `Object`) downto its private ones
- In dispatch tables, subclasses arrange its methods similarly, but whenever a method is shadowed, will dispatch on the one of the closest parent's (may be himself)

**Activations**

An **Activation** is an invocation of a procedure / function. Its **lifetime** lasts until the last step of execution of that procedure.

- For two different activations `a`, `b`, their lifetimes are either **Non-overlapping or Nested**
- An Activation is a particular instance of the function's invocation
- Sequence of function calls represented as an **Activation Tree**
  - e.g.
  ```plaintext
  Main
  /
  /  
  /    
  f     g
  / 
 /   
/    g
  ```
  - Earlier Activation goes on the left

**A Stack** can be used to track current Activations, which is a common practice in modern Languages. On each invocation, an **Activation Record** is pushed onto the stack. It is popped out when the procedure ends.

The design of Activation Records is an important part of the Runtime System, e.g.

- What needs to be inside an Activation Record
- Their exact layouts
- *Caller / Callee* is responsible for which part

**Runtime Errors**

The Code Generator usually assumes that the input IR is correct, since it has passed lexical, syntax & semantic error checkings. Therefore the generator will not check any errors. However, even those **type-safe** programs can fail to execute, due to **Runtime Errors**:

- Dispatch on `void`: design of Type System has flaws
- Division on zero: we can hardly know what is the exact dynamic value of a denominator at compile-time
- Case match failed on all branches
- ... 

We should generate codes which will make correct judgments and invoke corresponding run-time **exception** routines wherever there might be a Runtime Error.

**cgen** **For Pure Stack Machine**

The `cgen` Function is an abstraction of how a recursive Code Generator is implemented. `cgen(e1, n)` means emitting code for expression `e1`, when the current available temporary offset is `n`. Offsets only serve `let / case` expressions because they introduce new temporary variables.
Here we consider the generation of MIPS assembly code from AST structures. Each type of nodes on the input AST must have a corresponding implementation of `cgen`. We use a pure Stack Machine scheme to simplify the ideas, where:

- Only assuming 1 preserved Register - the Accumulator `$a0` to store:
  - Result of each operation (including function return value)
  - Self object pointer on method dispatch
- **Invariants**: The stack after each `cgen` will be exactly the same as at the point of entrance
- The stack is globally preserved, so usually using the memory as stack, and `$sp` for the stack pointer
- Use `$fp` for the frame pointer, the boundary of caller's and callee's responsibility

The following is a summary of implementations of recursive `cgen` function (without considering OOP):

<table>
<thead>
<tr>
<th>Expression</th>
<th>Implementation</th>
<th>Expression</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer <code>i</code></td>
<td><code>li $a0 $i</code></td>
<td><code>e1 + e2</code></td>
<td><code>cgen(e1)</code> <code>push $a0</code> <code>cgen(e2)</code> <code>add $a0 $t1 $a0</code> <code>pop</code></td>
</tr>
<tr>
<td><code>if e1 = e2 then e3 else e4</code></td>
<td><code>cgen(e1)</code> <code>push $a0</code> <code>cgen(e2)</code> <code>slt $t1 $a0</code> <code>pop</code> <code>beq</code> <code>$a0 $t1</code> <code>true_branch</code> <code>cgen(e3)</code> <code>j</code> <code>end_if</code> <code>false_branch:</code> <code>cgen(e4)</code> `` <code>end_if:</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>def f(x1, ..., xn) {e}</code></td>
<td><code>f_entry:</code> <code>move $fp $sp</code> <code>push $ra</code> <code>cgen(e)</code> <code> </code> <code>Sra $ra top</code> <code>addiu $sp $sp 4*$8</code> <code>lw $fp 0($sp)</code> <code>jr $ra</code> <code>f(e1, ..., en)</code></td>
<td><code>f(e1, ..., en)</code></td>
<td><code>push $fp</code> <code>cgen(en)</code> <code>push $a0</code> <code>...</code> <code>cgen(e1)</code> <code>push $a0</code> <code>j</code> <code>f_entry</code></td>
</tr>
<tr>
<td><code>let x : T &lt;- e1 in e2</code></td>
<td><code>cgen(e1, n)</code> <code>push $a0</code> <code>cgen(e2, n+1)</code> <code>pop</code></td>
<td>Temporary var <code>x</code> (whose offset is at <code>ofs</code>)</td>
<td><code>lw $a0</code> <code>-ofs($sp)</code></td>
</tr>
</tbody>
</table>

The offset `n` passed down the `cgen` function is used at `let` / `case` expressions, since they introduce new variables, and we need to save their values in inner scopes.

**Register Allocation**

Pure Stack Machines are simple but very inefficient. The most direct optimization is to use as much preserved registers (`$s0 `- `$s6` for MIPS) instead of always pushing onto stack. We need the following concepts for analyzing register allocation:

- **Next-Use** tells when will the value of `x` assigned at `x ← y + z (i)` be next used.
  - `= j` if the next closest usage is at `a op x (j)`.
- `x` is **Live** at some location when:
  1. It has been assigned a value previously
  2. It will be used after
  3. NO interleaving assignment to `x` between current location and the next usage

**Determine Liveness**

To determine the Liveness of variables in every location inside a **Basic Block**:
To determine the Liveness of variables throughout the Data Flow (i.e. across Basic Blocks), we should apply Dataflow Analysis framework, which will be covered in the last chapter.

**Register Interference Graph**

After determining Liveness of all variables, we can decide which register should be assigned to which variable. Basic idea is when two Temporaries \( a, b \) will live *simultaneously* at some point, called \( a \) interferes with \( b \), then they cannot share the same register.

A **Register Interference Graph (RIG)** is used to handle such a problem when we have in total \( k \) available registers, where:

- e.g.

![Register Interference Graph](image)

- Each node is a Temporary variable
- Each edge means an *interference* between nodes, and that these two nodes cannot share the same register

Finding a solution is a **Graph \( k \)-Coloring** problem, which is \( \text{NP-Hard} \). We use the following heuristic algorithm to partially solve this problem:

```java
void computeLiveness(set live_at_exit) {
    live_set = live_at_exit;
    for (each instruction i from end to start) {
        /* Suppose i is x <- y op z here. */
        live_set = live_set - {x};
        live_set = Union of live_set and {y, z};
        Liveness at location just before instruction i is live_set;
    }
}
```

```java
dict assignRegister(Graph RIG, set regs) {
    while (RIG is not empty) {
        if (there is a node n with < k neighbors)
            Push n onto stack;
        else { /* Run in short of registers. */
            Pick a victim node n;
            Spill n into memory;
        }
        Remove n from RIG;
    }
    for (each node n on stack) {
        Pick a reg \$rx\ from regs, which cannot be already used by one of n's neighbors;
        Assign \$rx\ to n;
    }
}
```

For a victim \( x \) spilled into memory, we need:

- **load** \( x \) every time before using
- **store** \( x \) every time after assignment

**Garbage Collection**

An object instance \( x \) is **Reachable** on heap iff some variable (either in register or in memory) points to \( x \), or another Reachable object \( y \) contains a pointer to \( x \). Unreachable objects are called **Garbage**, and is desired to get recycled by automatic memory management.

The concept of Reachability is *sound (safe)* but not complete, since Unreachable objects are definitely useless, but not all Reachable objects will be used later.
A example snapshot of the heap during execution can be:

- e.g.

```
$\begin{array}{cccccc}
\text{A} & \text{B} & \text{C} & \text{D} & \text{E} \\
\text{acc} & & & & \\
\end{array}$
```

- Arrows indicate reference pointings
- Roots include all references coming from outside the heap (in acc or on stack)

Various strategies of doing Garbage Collection (GC) exist. Three simple strategies are introduced below.

**Mark & Sweep**

When running out of memory conduct the following two stages:

1. Start from Roots, mark all Reachable objects
2. Erase all Unreachable objects, while leaving Reachable ones unmoved

Will fragment the memory, but no need to update pointers since unmoved.

**Stop & Copy**

Memory is partitioned into two equal areas $S_{\text{odd}}, S_{\text{new}}$, while $S_{\text{odd}}$ is the one under use currently. When $S_{\text{odd}}$ runs full, copy all Reachable objects to the beginning of $S_{\text{new}}$, and the rest of the memory is then considered free.

- e.g.

```
$\begin{array}{cccccccccccccc}
\text{root} & \text{A} & \text{B} & \text{C} & \text{D} & \text{E} \\
\end{array}$
```

- Notice the order:
  1. First copy a Root $A$
  2. Follow its out-going reference to $C$, copy $C$
  3. Update the pointer in $A$
  4. Repeat, starting from $C$
  5. If a referenced child is already copied, simply update the pointer

Avoids fragmentations, but is time- and memory-expensive, since pointers need to be updated, and only half of memory is available.

**Reference Counting**

Reference Counting (RC) is a dynamic GC strategy. We denote $rc(x)$ as the Reference Count of object $x$, where:

1. A new object $x$ has $rc(x) = 1$
2. After each assignment $x \leftarrow y$, $rc(x) = 1$, $rc(y) + 1$
3. When a variable $x$ (pointing to $x$) goes out of Scope, $rc(x) = 1$
4. Free $\emptyset$-referenced objects at certain times

Easy to implement, but very slow, and CANNOT handle circular references (where each $rc > 0$, but the whole group is not Reachable).

**Optimizations**

Optimization Schemes

Optimizations (优化) are conducted on IR:
There are three different Granularities of Optimizations, from less powerful (complex) to most powerful (complex):

1. **Local Optimizations** apply inside a Basic Block
2. **Global (Intra-procedural) Optimizations** apply to a CFG across Basic Blocks
3. **Inter-procedural Optimizations** apply across method boundaries

### Local Optimization Techniques

The following are 5 different Local Optimization techniques that can be applied to expressions inside a single Basic Block.

1. **Algebraic Simplification**: simplify obvious algebra calculations, e.g.
   - \( x := x + 0 \) / \( x := x * 1 \) \Rightarrow Deleted
   - \( x := x ^ 0 \) \Rightarrow x := 0
   - \( x := x ^ 2 \) \Rightarrow x := x + x (Only on machines where + is faster than *)
   - \( x := x ^ 8 \) \Rightarrow x := x \&< 3 (Only on machines where \&< is faster than *)

2. **Constant Folding**: compute constant expressions at compile time, e.g.
   - \( x := 1 + 2 \) \Rightarrow \( x := 3 \)
   - \( if 2 < 0 jump Label \) \Rightarrow \( if false jump Label \) \Rightarrow Deleted

3. **Dead Code Elimination**: remove codes that is meaningless, which
   - 1. Will never get executed, or
   - 2. Assigns to a Non-live Variable

4. **Common Subexpression Elimination**: replace common right-side expressions with previous assigned variable
   - e.g. \( b := a - d \) \( c := a - d \) \Rightarrow \( b := a - d \) \( c := b \)
   - MUST ensure that the assigned variable & everything in the expression is NOT changed between previous assignment and where replacement occurs
   - For SSA, the above property holds naturally

5. **Copy Propagation**: replace subsequent uses of copier variable with copiee
   - e.g. \( a := b \) \( x := 2 * a \) \Rightarrow \( a := b \) \( x := 2 * b \)
   - MUST ensure that the assigned variable & everything in the expression is NOT changed between previous assignment and where replacement occurs
   - For SSA, the above property holds naturally
   - NOT Optimization itself; only useful for triggering other Optimizations

To perform Local Optimizations, we combine the 5 techniques iteratively:

```java
void localOptimization() {
    do {
        Choose a technique and perform it;
    } while (still have possible Optimizations && iteration threshold not met);
}
```

### Global Optimizations

Similar to Local ones, there are several Global Optimization techniques which can be applied across basic blocks in a CFG.

1. **Global Common Subexpression Elimination**
2. **Global Copy Propagation**
   - CANNOT be simply applied to Array elements, because the Array might be modified somewhere else
3. **Code Motion**: move invariants outside of loop
4. **Induction Variables & Reduction in Strength**: simplify fixed patterns in loops, e.g.
   - \( j := j - 1 \) \( t4 := 4 * j \) \Rightarrow \( t4 := t4 - 4 \)
   - Need to handle following usages of \( j \) properly

Global Optimizations might trigger new possibilities of Local Optimizations, so we can iterate as follows:
Dataflow Analysis

Dataflow Analysis Abstraction

Global Optimizations and all the other analysis techniques which rely on the information across Basic Blocks require Dataflow Analysis. The main task is to collect needed information (e.g. Definitions) at certain point of the program Control Flow.

We use a mathematical framework called Dataflow Analysis Schema to handle such analysis. Suppose we have such a CFG:

- For each Statement $s$, define the following two Status of things we are interested in:
  - $in[s]$ describes the status before executing $s$
  - $out[s]$ describes the status after executing $s$
- For each Statement $s$, it also determines a Transfer Function $f_s$, where
  - $out[s] = f_s(in[s])$, i.e. describes the effect of executing $s$
  - Should be different for different sceneries
- For each Basic Block $B$, define the following two Status similarly:
  - $in[B]$ describes the status before entry of $B$
  - $out[B]$ describes the status after exit of $B$
- For each Basic Block $B$, it also determines a Transfer Function $f_B$, where
  - $out[B] = f_B(in[B])$, i.e. describes the effect of going through $B$
  - $f_B$ is a composition of $f_s$ for $s \in B$, e.g. $f_B = f_{s_2} \circ f_{s_1} \circ f_{s_0}$
- For each edge $B_0 \rightarrow B_e$ in the CFG, there are two possibilities:
  1. The endpoint is not a Join Node (e.g. the higher two edges in the example), then $in[B_e] = out[B_0]$
  2. The endpoint is a join Node who has predecessors $B_0, B_1, \ldots, B_n$, then
     $in[B_e] = out[B_0] \land out[B_1] \land \cdots \land out[B_n]$
     - Meet Operator $\land$ also depends on the problem scenery

With this standard framework, whenever we have a specific problem scenery, we can solve it with the following procedure:

1. Determine what should $in$ / $out$ / Transfer Function $f$ / Meet Operator $\land$ be
2. List relationships for $\forall$ Basic Block $B$:
   - $out[B] = f_B(in[B])$
   - $in[B] = \bigwedge out[\text{predecessors of } B]$
3. Initial conditions of $out[\text{entry}]$ or $in[\text{exit}]$ should be given
4. Iterate through all relationships until a Fixed Point Solution is met

Scenery: Reaching Definitions

A Definition $d$ Reaches a point $p$ if there is a path $d \rightarrow p$ such that $d$ is not overwritten. The problem of Reaching Definitions is one of the Dataflow Analysis sceneries, which can be stated as: "For each Basic Block in the program’s CFG, determine which definitions reach that point".

```java
void globalOptimization() {
   do {
      do {
         Choose a Local Optimization and perform it;
      } while (still have possible Local Optimizations);
      Choose a Global Optimization and perform it;
   } while (still have possible Optimizations & iteration threshold not met);
}
```
• \textit{in / out}: set of Definitions \( \{d_0, d_1, \ldots\} \)

• \(\text{out}[s] = f_s(\text{in}[s]) = \text{Gen}[s] \cup (\text{in}[s] - \text{Kill}[s])\)
  
  \begin{itemize}
    \item \(\text{Gen}[s]\) means the Definition \(d\) generated in \(s\) (if \(s\) is \(d: x = \ldots\))
    \item \(\text{Kill}[s]\) means set of all other Definitions of \(x\) in the program
  \end{itemize}

• \(\wedge\) is simply Union (\(\cup\))

An iterative algorithm can be:

\begin{verbatim}
void reachingDefinitions(Dataflow CFG) {
  for (each Basic Block \(B\) other than entry)
    out[B] = \{\};
  do {
    for (each Basic Block \(B\) other than entry) {
      in[B] = Meet of all out[predecessor of \(B\)];
      out[B] = f_B(in[B]);
    }
  } while (any changes occur to any out[B] set);
}
\end{verbatim}

To save space and accelerate the algorithm, we can use a Bitmap (Bit-vector) to represent \(\text{in}[B] / \text{out}[B]\) sets.

\section*{Scenery: Liveness Analysis}

A Variable \(v\) is \textbf{Live} at point \(p\) iff it has been defined now and will be used along some path in the CFG starting at \(p\). Otherwise \(v\) is \textbf{Dead} and that can trigger Dead Code Elimination. The problem of \textbf{Liveness Analysis} can be stated as:

"For each Basic Block in the program's CFG, determine which variables are Live at that point".

Note that Liveness Analysis is conducted backward along the CFG edges, therefore the framework is slightly different:

• Initial condition should be \(\text{in}[\text{exit}]\)
• Transfer Function reversed, i.e. \(\text{in}[B] = f_B(\text{out}[B])\)
• Meet Operations occur at startpoints of edges

• \textit{in / out}: set of Live Variables \(\{v_0, v_1, \ldots\}\)

• \(\text{in}[s] = f_s(\text{out}[s]) = \text{Use}[s] \cup (\text{out}[s] - \text{Def}[s])\)
  
  \begin{itemize}
    \item \(\text{Use}[s]\) means set of all Variables \(\{y, z\}\) used at \(s\) (if \(s\) is \(x = y + z\))
    \item \(\text{Def}[s]\) means the Variable defined at \(s\) (\(x\))
  \end{itemize}

• \(\wedge\) is simply Union (\(\cup\))

An iterative algorithm can be:

\begin{verbatim}
void livenessAnalysis(Dataflow CFG) {
  for (each Basic Block \(B\) other than exit)
    in[B] = \{\};
  do {
    for (each Basic Block \(B\) other than exit) {
      out[B] = Meet of all out[successor of \(B\)];
      in[B] = f_B(out[B]);
    }
  } while (any changes occur to any in[B] set);
}
\end{verbatim}

\section*{Scenery: "Must-reach" Definitions}

A Definition \(d\) "Must-reach" a point \(p\) iff \(\forall\) paths \(\rightarrow p\), \(d\) appears at least once and will not be overwritten. In this case:

• \(\wedge\) should be \(\cap\)
• All other setups are the same as Reaching Definitions

\section*{Semi-Lattice Diagram}

Dataflow Analysis framework can be represented as a mathematical \textbf{Meet Semi-Lattice (最大下界半格)} Diagram. That Semi-Lattice is a \textbf{Partially-ordered (偏序的)} set which has a \textbf{Greatest Lower Bound (i.e. Meet)} for \(\forall\) finite subset.
- **Domain**: $V$ of the problem is the set of all possible values (e.g., set of all definitions).
- Greatest Lower Bound of subset $x$ and $y = x \wedge y =$ first common successor of $x$ & $y$.
- A partial-order $x \leq y$ indicates there is a path $y \rightarrow x$.
  - If Meet Operation is $\sqcap$, largest subset (i.e., $\text{Top}$) is $\emptyset$, and smallest subset (i.e., $\text{Bottom}$ $\sqcup$) is the whole Domain.
  - If Meet Operation is $\sqcap$, then largest is the whole Domain, and smallest is $\emptyset$.

Meet Operator follows several properties:

1. **Idempotent**: $x \wedge x = x$
2. **Commutative**: $x \wedge y = y \wedge x$
3. **Associative**: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$

Partial-order should have several properties (similar to Equivalent relations, except for Anti-symmetric):

1. **Reflexive**: $x \leq x$
2. **Anti-symmetric** (反对称): if $x \leq y$ and $y \leq x$ then $x = y$
3. **Transitive** (传递): if $x \leq y$ and $y \leq z$ then $x \leq z$

For a Dataflow Analysis framework $(F, V, \wedge)$ with Transfer Functions family $F$:

- It is **Finite-descending** iff every descending chain from Top to Bottom has finite length.
- It is **Monotone** (单调的) iff $x \leq y \Rightarrow f(x) \leq f(y)$
- It is **Distributive** (可分配的) iff $f(x \wedge y) = f(x) \wedge f(y)$ (this is a special case of Monotonicity)

### Scenery: Constant Propagation

The problem of **Constant Propagation** can be stated as: “For each Basic Block in the program’s CFG, determine which variables are Constant and their Values at that point”.

- **Domain**: mappings from all Variables to its Value $\{(x, v_x), (y, v_y), \ldots\}$
  - $v_x$ can be either $\text{Undef} / \text{NAC}$ (NOT a Constant) / Constant $c$
- **Transfer Function** $f$ is defined as:
  - For non-assignment statement $s$, $f_s$ is an identity function.
  - For assignment statement $s : x = e$, $f_s$ produces new $v'_x$ where
    - If $e$ is Constant $c$, then $v'_x = c$
    - If $e$ is $y$ op $z$ and any of them is $\text{NAC}$, then $v'_x = \text{NAC}$
    - If $e$ is $y$ op $z$ and $v_y = c_1$, $v_z = c_2$, then $v'_x = c_1$ op $c_2$
    - If $e$ is $y$ op $z$, none of them is $\text{NAC}$ and any of them is $\text{Undef}$, then $v'_x = \text{Undef}$
    - Else (e.g., $e$ is a function call), $v'_x = \text{NAC}$
- **Meet Operation** $v_x \wedge v_y$ is defined as:
  - If any of them is $\text{NAC}$, then $v_x \wedge v_y = \text{NAC}$
  - If any of them is $\text{Undef}$, then $v_x \wedge v_y =$ value of another one
  - If $v_x = c_1$, $v_y = c_2$ where $c_1 \neq c_2$, then $v_x \wedge v_y = \text{NAC}$
  - If $v_x = v_y = c$, then $v_x \wedge v_y = \text{Constant} c$

Under this scenery, the Meet Semi-Lattice framework is Monotone but NOT Distributive.