if() Computer Language & Compilers

> Author: Jose 胡冠洲 @ ShanghaiTech

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`

Notation	Meaning	Notes
st	Concatenation of s and t	$s\varepsilon = \varepsilon s = s$
s^n	n times self-concatenations	$s^0 = \varepsilon$
$\|\|s\|\|$	Length of s	

A **Language** L is a set of Strings over a fixed **Alphabet** Σ, constructed using a specific Grammar.

- e.g. $\{\varepsilon, 0, 01, 011, 0111, \ldots\}$
- 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

Notation	Meaning	Notes
$\emptyset$	Empty Language	$\neq \{\varepsilon\}$
$L \cup M$	Union of L and M	$\{s : s \in L \vee s \in M\}$
$L \cap M$	Intersection of L and M	$\{s : s \in L \wedge s \in M\}$
LM	Set of all possible concatenation results	$\{st : s \in L \wedge t \in M\}$
L^*	Zero or more self-concatenations	L^+: One or more

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**

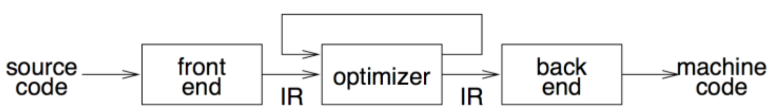
Semantic Analysis (语义分析) checks semantic errors, and generates IR.

- 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.

- e.g. INT, IDENTIFIER, WHILE, $\ldots$
- Each Token is a Language over the source program Alphabet, described by a certain RE
- The 1st layer of *abstraction*, which extracts the information of word elements

A **Lexeme** (词素) 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, 正则表达式)** 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 ...
ε	Put an empty String here	$L(\varepsilon) = \{\text{`'}\}$
a	Put a character `a` here	$L(a) = \{\text{`a'}\}$
$r_1 + r_2$	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).

Notation	Meaning	Notes
$[a-zA-Z]$	Anyone in range $[a, z]$ or $[A, Z]$	$= a + \cdots + Z$
r^+	*Positive Closure* of what r generates	$= r(r)^*$
$r?$	What r generates appear once or not	$= r + \varepsilon$
r^i	What r generates appear i times	$= rr\cdots r$, i times
$.$	Any single char in the whole Alphabet	

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

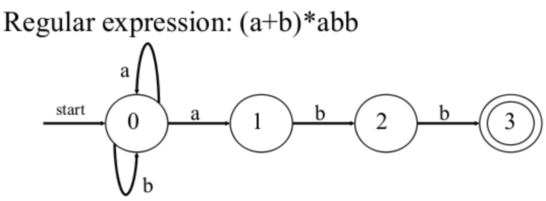
- e.g. For describing integers:
  - Digit $= [0-9]$
  - Integer $=$ Digit Digit *

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

- *Transition Edge*: from the previous state, given the next input char, will go to the next state
- *Accepting State*: marked with concentric circles; when ends at such state, we accept
- *Death State*: the error state trap; all undefined transitions point to this state by default

- *Transition Table*:

	input	
state	a	b
0	{ 0, 1 }	{ 0 }
1	--	{ 2 }
2	--	{ 3 }

NFA

A **Non-deterministic Finite Automaton (NFA)** can have more than one alternative actions for the same input Symbol at the same State, and can have ε-Transitions (without consuming any input).

- Accepts s iff: there exist AT LEAST one path from Start State $\rightarrow$ an Accepting State that spells out s
- May have different behaviors for the same input stream

Notations	Meaning
s	A State
S	Finite set of States
s_0	Start (Initial) State
F	Set of all Accepting States
`move (S, c)`	Function returning set of all possible States that $\forall s \in S$ can goto with input `c`
`eps-closure (S)`	Function returning the ε-Closure of set S

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

DFA

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

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

Notations	Meaning
`move (s, c)`	Function returning the next state that s goes to with input `c`

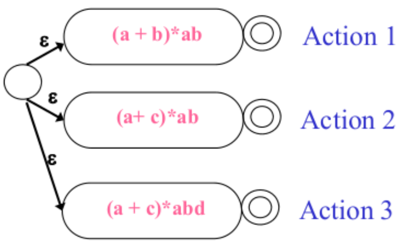
> No ε-Closure concept for DFAs.

Implementation of Lexers

Each Token (described by a unique RE r) requires a unique *Recognizer*.

1. **[WAY 1]**: RE $r \rightharpoonup$ NFA $\rightharpoonup$ Recognizer
2. **[WAY 2]**: RE $r \rightharpoonup$ NFA $\rightharpoonup$ DFA $\rightharpoonup$ Recognizer
3. **[WAY 3]**: RE $r \rightharpoonup$ DFA $\rightharpoonup$ Recognizer
4. **[WAY 4]**: RE $r \rightsquigarrow$ DFA $\rightharpoonup$ *Minimized* DFA $\rightharpoonup$ Recognizer

The Lexical Analyzer is then built from a bunch of Recognizers:

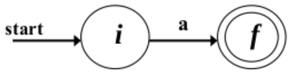


- Each Recognizer works for one Token
- Try in listed order, therefore ordering of Recognizers matters

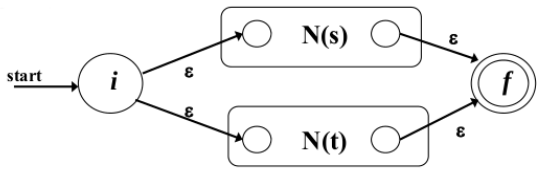
From RE $\rightarrow$ NFA

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

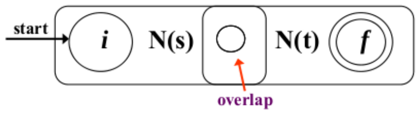
1. For ε / each $a \in \Sigma$:



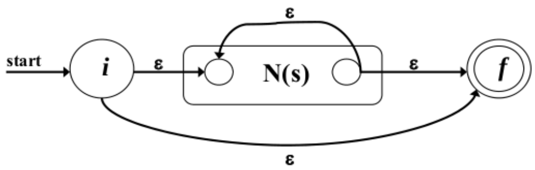
2. For $s + t$:



3. For st:



4. For s^*:



There are some requirements on such construction:

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

Properties of the resulting NFA:

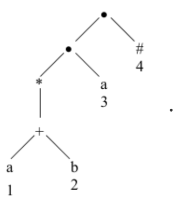
- Exactly 1 Start State & 1 Accepting State
- # of States in NFA $\leq 2\times$ (# of Symbols + # of Operators) in r
- States do not have multiple outgoing edges with the same input symbol
- States have at most 2 outgoing ε edges

From RE $\rightarrow$ 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:



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

[Step 3]: Compute `nullable()`, `firstpos()` & `lastpos()` for ALL nodes.

1. `firstpos(n)`: Function returning the set of positions where the *first* Symbol can be at, *in the sub-RE* rooted at `n`
2. `lastpos(n)`: Function returning the set of Positions where the *last* Symbol can be at, *in the sub-RE* rooted at `n`
3. `nullable(n)`: Function judging whether *the sub-RE* rooted at `n` can generate ε

n	nullable(n)	firstpos(n)	lastpos(n)
ε-leaf	True	$\emptyset$	$\emptyset$
leaf at Position i	False	$\{i\}$	$\{i\}$
$c_1 + c_2$	nullable(c1) || nullable(c2)	firstpos(c1) $\cup$ firstpos(c2)	lastpos(c1) $\cup$ lastpos(c2)
$c_1 \cdot c_2$	nullable(c1) && nullable(c2)	nullable(c1) ? firstpos(c1) $\cup$ firstpos(c2) : firstpos(c1)	nullable(c2) ? lastpos(c1) $\cup$ lastpos(c2) : lastpos(c2)
c^*	True	firstpos(c)	lastpos(c)

[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 oprations when leaving $\cdot$ / * nodes:

- $c_1 \cdot c_2$: For all $i \in$ `lastpos(c1)`, `followpos(i)` = `followpos(i)` $\cup$ `firstpos(c2)`
- c^*: For all $i \in$ `lastpos(c)`, `followpos(i)` = `followpos(i)` $\cup$ `firstpos(c)`

[Step 5]: Construct the DFA.

```
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 $\in S$
- Start State of the resulting DFA is S_0

Calculate ε-Closure

Similar problem as *graph traversal*.

```
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 - $O(|r|)$; Time - $O(|r| \times |s|)$

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 - $O(2^{|r|})$; Time - $O(|s|)$

Convert NFA $\longrightarrow$ 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 S in resulting DFA is an Accepting State iff $\exists s \in S, s$ 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;
```

```
            }
            Split G according to the above information;
        }
    } while (PI changed in this iteration);
    Every Group in PI is a state in the minimal DFA;
}
```

- 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 $/ \Rightarrow \varepsilon$ 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.

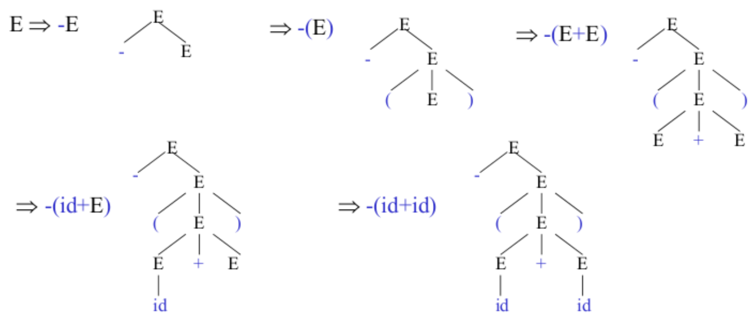
- *Hash Table* is suitable for this task
- Lexeme's position in source file (e.g. *line number*) is an important information for error handling

Syntax Analysis

Parse Tree Abstraction

A **Parse Tree / Symtax 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.

Notation	Meaning	Notes
$A \to \alpha$	A can be replaced with α in a step	Called a Production Rule
$A \to \alpha B \mid \beta$	Merges two rules starting from the same Non-terminal	
$A \Rightarrow s$	From Start Symbol A, by a Production Rule, we can derive s	Called a *Derivation Step*
$A \Rightarrow^* s$	From Start Symbol A, after Zero or more steps, can reach s	$\Rightarrow^+$ means One or more

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. $A \to \alpha A \mid \beta$, where A is a Non-terminal and α, β 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 β
- Corresponds to *Top Down Parsing*

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

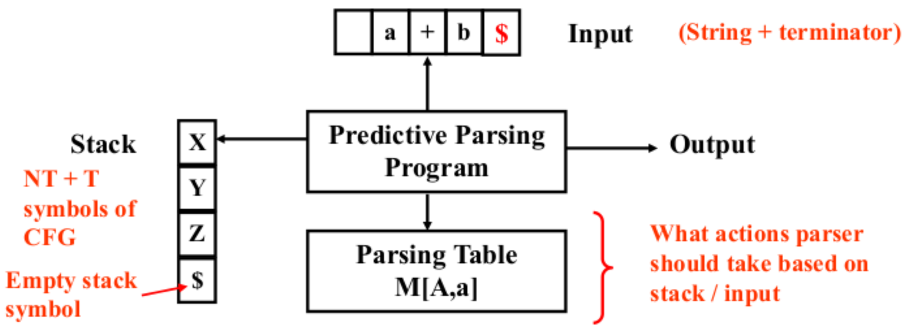
- If $\beta A \gamma \Rightarrow_{rm} \beta \delta \gamma$, then NO Non-terminals in γ
- Corresponds to *Bottom Up Parsing*, in reversed manner

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.



1. **[WAY 1]**: Eliminate Left Recursion $\rightharpoonup$ Recursive-descent Parsing
2. **[WAY 2]**: Eliminate Left Recursion $\rightharpoonup$ Left Factoring $\rightharpoonup$ Recursive Predictive Parsing
3. **[WAY 3]**: Eliminate Left Recursion $\rightharpoonup$ Left Factoring $\rightharpoonup$ Construct Parsing Table $\rightharpoonup$ 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:

- $A \to A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
- $\Downarrow$
- $A \to \beta_1 A' \mid \beta_2 A' \dots \mid \beta_n A'$
- $A' \to \alpha_1 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$

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 Prodcution 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 Pruduction Rule to expand now.

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

- $A \to \alpha\beta_1 \mid \dots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_m$
- $\Downarrow$
- $A \to \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_m$
- $A' \to \beta_1 \mid \dots \mid \beta_n$

> $\|LL(1)\| < \|$ 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;
```

```
                  B();
                  match current Token with 'e', move to next Token;
                  break;
        case 'c': match current Token with 'c', move to next Token;
                  B();
                  match current Token with 'd', move to next Token;
                  break;
        case 'f': C();              /* Since 'f' in FIRST(C). */
                  break;
        default:  raise ERROR;
    }
}
void B() {
    switch (current Token) {
        case 'b': match current Token with 'b', move to next Token;
                  B();
                  break;
        case 'e':
        case 'd': nothing;          /* Since 'e'/'d' in FOLLOW(B). */
                  break;
        default:  raise ERROR;
    }
}
void C() {
    switch (current Token) {
        case 'f': match current Token with 'b', move to next Token;
                  break;
        default:  raise ERROR;
    }
}
```

Parsing Table Construction

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.

- e.g.

Non-terminal	Input symbol					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

[Step 1]: Compute `FIRST()` for every Terminal and Non-terminal.

```
void computeFirst() {
    Initialize all FIRST() to be an empty set;
    for (every Terminal t)
        FIRST(t) is assigned to {t};
    do {
        for (every Production Rule r: X -> ...) {
            if (r is X -> epsilon)
                Add 'eps' into FIRST(X);
            else {
                /* Suppose r is X -> Y1 Y2 ... Yk. */
                for (i from 1 to k) {
                    FIRST(X) = Union of FIRST(X) and FIRST(Yi);
                    if (epsilon is not in FIRST(Yi))
                        break;
                }
            }
        }
    } while (there are updates in this iteration);
```

```
}
```

- Checking "$X \Rightarrow^* \varepsilon$?" is equivalent to Checking "$\varepsilon \in$ `FIRST(X)` ?"
- `FIRST(X1, X2, ..., Xk)` represents `FIRST()` for the stream `X1 X2 ... Xk`
  - e.g. If `X1` and `X2` may be ε, but `X3` cannot, then
  - `FIRST(X1, X2, ..., Xk)` $=$ `FIRST(X1)` $\cup$ `FIRST (X2)` $\cup$ `FIRST (X3)`

[Step 2]: Compute `FOLLOW ()` for every Non-terminal.

```
void computeFollow() {
    Initialize all FOLLOW() to be an empty set;
    Add "$" into FOLLOW(Start Symbol S);
    do {
        for (every Production Rule r: X -> Y1 Y2 ... Yk) {
            for (i from 1 to k) {
                if (Yi is a Non-terminal) {
                    FOLLOW(Yi) = Union of FOLLOW(Yi) and (FIRST(Yi+1, Yi+2, ... Yk) - {epsilon});
                    if (i == k || epsilon is in FIRST(Yj+1, Yj+2, ... Yk))
                        FOLLOW(Yi) = Union of FOLLOW(Yi) and FOLLOW(X);
                }
            }
        }
    } while (there are updates in this iteration);
}
```

[Step 3]: Build the Parsing Table.

```
void buildParsingTable() {
    for (every Production Rule r: X -> Y1 Y2 ... Yk) {
        for (every possible Terminal t) {
            if (t is in FIRST(Y1, Y2, ..., Yk))
                Add r into Table[X, t];
        }
        if (epsilon is in FIRST(Y1, Y2, ..., Yk)) {
            for (each terminal b in 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:
>
> - $S \rightarrow iCtSE \mid a$
> - $E \rightarrow eS \mid \varepsilon$
> - $C \rightarrow b$

Implementing $LL(1)$ Parsing

Non-recursive Parsing / $LL(1)$ **Parsing** uses a *stack* instead of *recursions*, which is more efficient, but needs a correct Parsing Table (*Table-driven*).

```
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. */
```

```
            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;
        }
    }
}
```

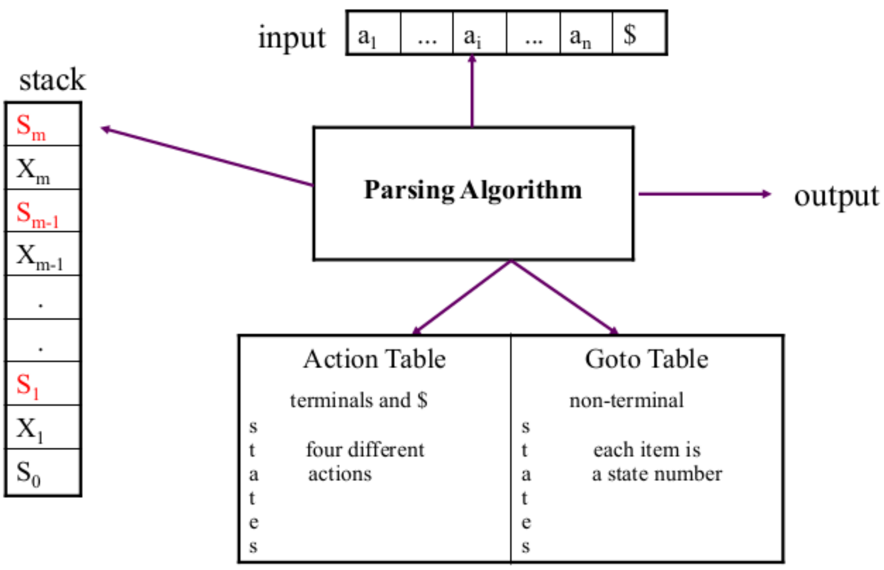
- Example procedure of $LL(1)$ Parsing:

stack	input	output
\$S	abba\$	S → aBa
\$aBa	abba\$	
\$aB	bba\$	B → bB
\$aBb	bba\$	
\$aB	ba\$	B → bB
\$aBb	ba\$	
\$aB	a\$	B → ε
\$a	a\$	
\$	\$	accept, successful completion

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 γ 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 $\rightharpoonup$ $LR(0)$ Action & Goto Table $\rightharpoonup$ Parser
2. **[WAY 2]**: $LR(0)$ Automata $\rightharpoonup$ $SLR(1)$ Action & Goto Table $\rightharpoonup$ Parser
3. **[WAY 3]**: $LR(1)$ Automata $\rightharpoonup$ $LR(1)$ Action & Goto Table $\rightharpoonup$ Parser
4. **[WAY 4]**: $LALR(1)$ Automata $\rightharpoonup$ $LALR(1)$ Action & Goto Table $\rightharpoonup$ 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 $A \to s$, 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 $A \to \beta_1\beta_2$ is **Valid** for Viable Prefix $\alpha\beta_1$ iff $S \Rightarrow^* \alpha A\gamma \Rightarrow \alpha\beta_1\beta_2\gamma$
  - If $\beta_2 = \varepsilon$, should Reduce
  - If $\beta_2 \neq \varepsilon$, should Shift

An $LR(0)$ **Item** $A \to \beta_1 . \beta_2$ means that:

- Production Rule $A \to \beta_1\beta_2$ is Valid for current Viable Prefix
- We have shifted things in β_1 onto stack, but things in β_2 not met yet
- No information about next Tokens, i.e. no Look-aheads

[Step 1]: Define `CLOSURE()` to decide States.

```
set computeClosure(set I) {
    closure = I;
    do {
        for (every Item m in I) {
            /* Suppose m is A -> a.Bb 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 `GOTO()` to decide Transitions.

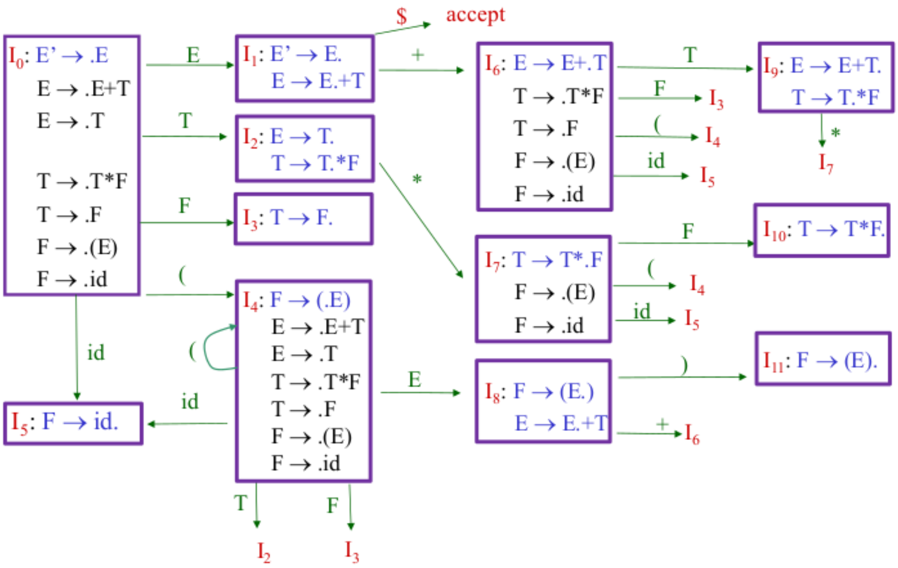
```
set computeGoto(set I, Symbol X) {
    result = {};
    for (every Item m in I) {
        /* Suppose m is A -> a.Xb here. */
        result = Union of result and CLOSURE({A -> aX.b});
    }
    return result;
}
```

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

```
void buildLR0Automata() {
    I0 = CLOSURE({S' -> .S});
    DStates = {I0};
    do {
        for (each Item set I in DStates) {
            for (each Grammar Symbol X) {
                J = 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 $\forall$ State I containing $S' \rightarrow S.$, `GOTO(I, $) = 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 $(I_0 X_1 I_1 \dots X_m I_m, a_i a_{i+1} \dots a_n \$)$, where:

- $I_0 X_1 I_1 \dots X_m I_m$ is current Stack content, *bottom* to *top*
- $a_i a_{i+1} \dots a_n\$$ is the rest of the input Token stream
- Represents:
    - A *snapshot* at some time in the Parsing process
    - A Right-most Derivation $S \Rightarrow^* X_1 \dots X_m a_i a_{i+1} \dots a_n \$$

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

```
/* 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 $\Rightarrow$ Multiple Actions in 1 Action Table entry; If *no Conflicts happen*, then G is a $LR(0)$ Grammar
- Example of an Action & Goto Table:

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

	LR(0) Action Table							Goto Table		
state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2	r2	r2	s7 r2	r2	r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9	r1	r1	s7 r1	r1	r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

  - `s2` - Shift to State I_2
  - `r3` - Reduce by Production Rule #3
- Example procedure of $LR(0)$ Parsing:

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	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.

```
/* 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. && a is in FOLLOW(A))
                    Add "reduce A -> 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 `CLOSURE()` to decide States.

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

[Step 2]: Define `GOTO ()` to decide Transitions.

```
set computeGoto(set I, Symbol X) {
    result = {};
    for (every Item m in I) {
        /* Suppose m is A -> a.Xb, x here. */
        result = Union of result and CLOSURE({A -> aX.b, x});
    }
}
```

[Step 3]: Build $LR(1)$ Automaton. The dummy item here is $S' \to .S$, \$.

- Shorthand for $r, a_1; r, a_2; \ldots; r, a_n$ is $r, a_1/a_2/\ldots/a_n$
- A State will contain $A \to \alpha., a_1/a_2/\ldots/a_n$, where $\{a_1, a_2, \ldots, a_n\} \subseteq$ `FOLLOW (A)`

Implementing $LR(1)$ Parsing

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

```
/* 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, x)      /* Shift is not effected. */
                    Add "shift GOTO(i, a)" in Action[i, a];
                else if (r is A -> D., a)   /* Reduce only when match. */
                    Add "reduce A -> D" in Action[i, a];
                else if (r is S' -> S., "$")
                    Add "ACCEPT" in Action[i, "$"];
            }
        }
    }
}
```

- May still leave Conflicts; If *no Conflicts happen*, then G is a $LR(1)$ Grammar

Build $LALR(1)$ Automata

A **Core** is the set of all $LR(0)$ Items in a $LR(1)$ State, ignoring the following Terminal symbol.

$LALR(1)$ *merges* all the $LR(1)$ states with the same *Core*.

- Is a *Trade-off* between Grammar range ($LR(1)$) v.s. Efficiency ($SLR(1)$)
    - Number of States in $LALR(1)$ Automata $=$ Number of States in $SLR(1)$ Automata
    - Will only introduce *Reduce / Reduce Conflict*s into original $LR(1)$ Parser; If *no Conflicts happen*, then G is a $LALR(1)$ 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

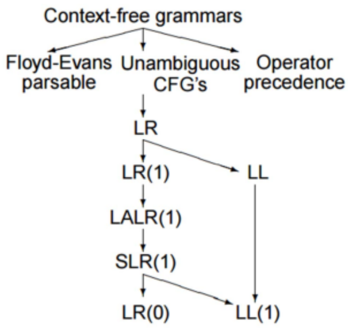
NOT Context-free Language $=$ CANNOT write a CFG for this Language.

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

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:



Error Handling

Types of Errors

Error Type	Example	Detector
Lexical	`x # y = 1`	Lexer
Syntax	`x = 1 y = 2`	Parser
Semantic	`int x; y = x(1)`	Type Checker
Correctness	Can compile, but wrong output	User / Static Analysis / Model Checker / $\cdots$

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` $\Rightarrow$ 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` $\Rightarrow$ 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 $E_1 \to ID := Expr$ 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:

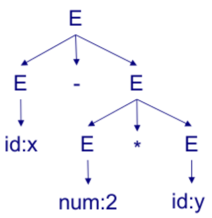
- Structural
  - **Abstract Syntax Trees** (*AST*)
  - **Directed Acyclic Graphs** (*DAG*)
  - **Control Flow Graphs** (*CFG*)
  - **Data Dependence Graphs** (*DDG*)
- Linear
  - **Static Single Assignment Form** (*SSA*)
  - **3-address Code**
  - **Stack Code**

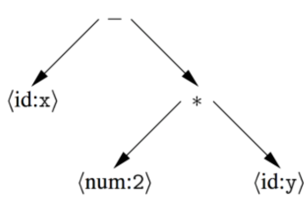
There will be hybrid combinations, and which to choose strongly depends on the design goals of the compiler system.

Abstract Syntax Tree

AST is a simplified Parse Tree.

Example:





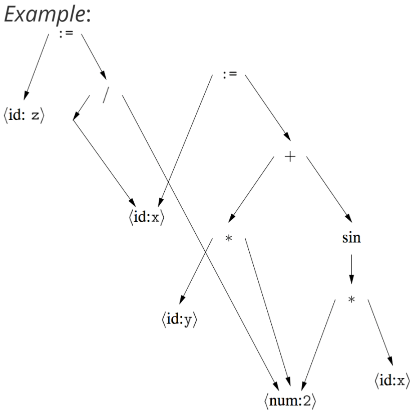
- Advantages
  - Close to source code
  - Suitable for source-source translation

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

Directed Acyclic Graph

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

Example:



- 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. 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 $S_1, \dots, S_n$, where flow must enter this block only at S_1, AND if S_1 is executed, then $S_2, \dots, S_n$ are executed strictly in that order, unless one Statement causes halting.

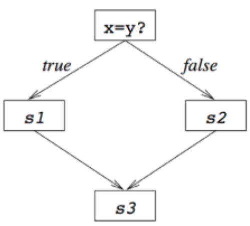
- The **Leader** is the first Statement of a Basic Block
- A **Maximal Basic Block** is a maximal-length Basic Block

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

- $\exists$ edge $b_1 \to b_2$ 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
S3
```



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;
```

➔

```
x_1 := 3;
x_2 := x_1 + 1;
x_3 := 7;
x_4 := x_3*2;
```

A ϕ**-function** generates an extra assignment to "choose" from Branches or Loops. If Basic Block B has *Predecessors* $P_1, \dots, P_n$, then $X = \phi(v_1, \dots, v_n)$ assigns $X = v_j$ if control enters B from P_j.

- e.g.
  1. 2-way Branch:

```
if (...)                 if (...)
    X = 5;                   X_0 = 5;
else                     else
    X = 3;                   X_1 = 3;
                         X_2 = φ(X_0, X_1);
Y = X;                   Y_0 = X_2;
```

  2. While Loop:

```
    j = 1;                   j_5 = 1;
s: // while (j < x)      s:     j_2 = φ(j_5, j_4);
    if (j >= X)              if (j_2 >= X)
        goto E;                  goto E;
    j = j+1;                 j_4 = j_2+1;
    goto s                   goto s
E:                       E:
    N = j;                   N = j_2;
```

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

Where to place a ϕ?

- If Basic Block B contains an assignment to variable X, then a ϕ MUST be inserted before each Basic Block Z that:
  1. $\exists$ non-empty path $B \to^+ Z$
  2. $\exists$ 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:

assignments	`x = y op z`
	`x = op y`
	`x = y[i]`
	`x = y`
branches	`goto L`
conditional branches	`if x relop y goto L`

procedure calls	`param x`
	`param y`
	`call p`

address and pointer assignments	`x = &y` `*y = z`

- **Quadruples** (四元组):

x - 2 * y				
(1)	load	t1	y	
(2)	loadi	t2	2	
(3)	mult	t3	t2	t1
(4)	load	t4	x	
(5)	sub	t5	t4	t3

  - Uses explicit names to store results
  - Easy to reorder, but needs more fields
- **Triples** (三元组):

x - 2 * y			
(1)	load	y	
(2)	loadi	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

  - 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* informations 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 $A.syn$ are synthesized using αs' (children's) Attributes

- e.g. $A \to \alpha_1 + \alpha_2$, $A.val = \alpha_1.val + \alpha_2.val$
- A's Attribute val is synthesized from children's vals

Inherited Attributes like $\alpha_1.in$ are inherited (passed down) from A's (parent's) Attributes

- e.g. $L \to L_1, id$, $L_1.type = L.type$
- L_1's Attribute $type$ is inherited from L's $type$

Syntax-Directed Definitions

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

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

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 $c_1, c_2, \ldots, c_n$.

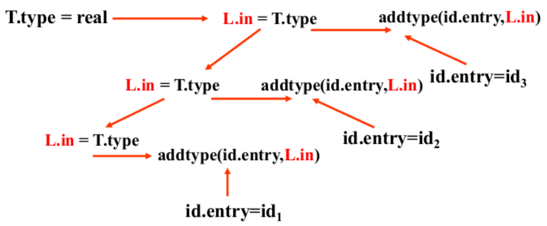
- This Semantic Rule must be evaluated AFTER Rules for $c_1, c_2, \ldots, c_n$
- Dependency can be represented by a directed **Dependency Graph**
  1. Mark the AST with Semantic Rules

2. Each Semantic Rule gets an id
3. Draw dependency relations between Rules
4. Verify that it is *Acyclic*

- e.g.

Production	**Semantic Rules**
D → T L	L.in = T.type
T → int	T.type = integer
T → real	T.type = real
L → L_1 , id	L_1.in = L.in, addtype(id.entry,L.in)
L → id	addtype(id.entry,L.in)

Evaluation **Input: real id_1, id_2, id_3**



S-Attributed Definitions only use Synthesized Attributes.

L-Attributed Definitions require that in each Production Rule $A \to \alpha_1\alpha_2\dots$ with Semantic Rule $b \to f(c_1, c_2, \dots, c_n)$:

- b is a Synthesized Attribute of A, OR
- b is an Inherited Attribute of α_j, which depends no more than Attributes of $A, \alpha_1, \dots, \alpha_{j-1}$

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 val when shifting by a valued Token (e.g. `int, 3`)
  - Push in a `_` (占位符) when shifting by a 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 α_j, put the action just before α_j
- e.g.

D → T { L.in = T.type } L
T → int { T.type = integer }
T → real { T.type = real }
L → id { addtype(id.entry,L.in), L_1.in = L.in } L_1
L → ε

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(A1.a, Y.y)}`
- $A \to X$ `{A.a = f(X.x)}`
- $\Downarrow$
- $A \to X$ `{A'.in = f(X.x)}` A' `{A.a = A'.syn}`
- $A' \to Y$ `{A1'.in = g(A'.in, Y.y)}` A'_1 `{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.

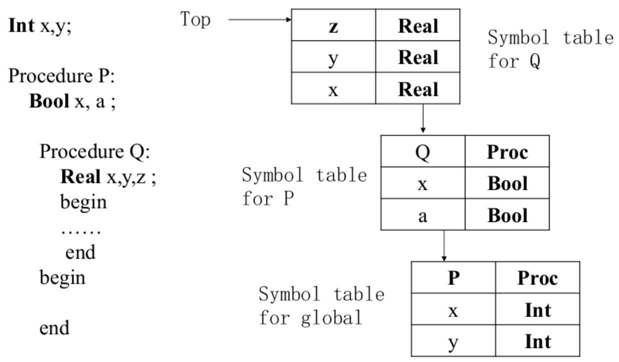
- May obey **Closest Enclosing Definition**
  - Can be nested
  - Refer to closest parent definition
- May obey **Globally Visible Definition**
  - CANNOT be nested
  - Can be used before defined

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.



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 $\dfrac{\vdash \text{Hypothesis}_1 \ldots \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$ to represent $H_1 \wedge \cdots \wedge H_n \Rightarrow Conclusion$, when each Hypothesis H and the Conclusion are in the form $Context \vdash expr : T$.

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. $O(x) = Int$
  - 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 `let` / `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 `SELF_TYPE` class, needed for handling `SELF_TYPE`s
  - Means we are inside Class C now
  - Properties:
    - `SELF_TYPE` $_C \leq C$
    - `SELF_TYPE` $_C \leq T$ if $C \leq T$
    - $lub($ `SELF_TYPE` $_C, T) = lub(C, T)$

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*: $\forall E, dynamicType(E) \leq staticType(E)$, where:
    - **Dynamic Type** is the run-time evaluated type of an Expression
    - **Static Type** captures all possible Dynamic Types
- **Least Upper Bounds**: $lub(T_1, \ldots, T_n)$ means the smallest parent class of all $T_1, \ldots, T_n$
  - Needed for `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.

$$\dfrac{\dfrac{\vdash \text{false:Bool}}{\vdash \text{not false: Bool}} \qquad \dfrac{\vdash \text{1:Int} \qquad \dfrac{\vdash \text{2:Int} \qquad \vdash \text{3:Int}}{\vdash \text{2*3: Int}}}{\vdash \text{1+2*3: Int}}}{\vdash \text{while not false loop 1 + 2 * 3 pool:Object}}$$

> 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** (操作语义), 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. $S = [l_x : 2, l_y : 0]$
  - $S[v/l_x]$ means to *update* S by adding information $S(l_x) = v$
    - Needed for `let` / `case` expressions, since they introduce new variables in new sub-scopes
  - A Rule may have *side effects*: change the Store
- **Self-object** so: current `self` object, needed for inferring `self`
  - Will never change after an operation

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

- T is the Dynamic Type of value v
- a_i is the ith Attribute, where the location of a_i's value is l_i
- Special notations for basic classes:
  1. $Int(5)$: integer value `5`
  2. $Bool(true)$: boolean value `true`
  3. $String(4,$ "Cool"$)$: string `"Cool"` with length `4`
  4. $void$: special instance of all types, only effective for `isvoid`

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

- $l_{new} = newloc(S)$ means allocate a new, free location l_{new} in memory
  - Needed for `let` / `case` / `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
- $class(T) = (a_1 : T_1 \leftarrow e_1, \dots, a_n : T_n \leftarrow e_n)$ illustrates the composition of Type T
  - Needed for `new` expressions
- $impl(T, f) = (e_1, \dots, e_n, e_{body})$ 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 (标记语义)
> - Axiomatic Semantics (公理语义)

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.

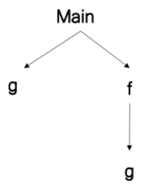
> For the detailed COOL Runtime System Conventions, refer to COOL Runtime System, section 2-5.

> - 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.

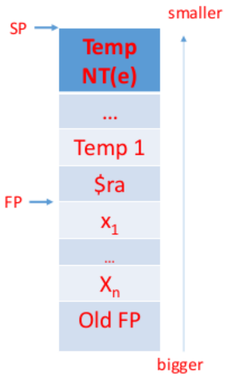
- For two different activations a, b, their lifetimes are either *Non-overlapping* or *Nested*
- An Activation is a particular instance of the function's invocation
- Sequence of function calls represented as an **Activation Tree**
  - e.g.



  - 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 (除零错误): we can hardly know what is the exact dynamic value of a *denominator* at compile-time
- Case match failed on all branches
- $\dots$

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>
<tr><th>Expression</th><th>Implementation</th><th>Expression</th><th>Implementation</th></tr>
<tr><td>Integer <code>i</code></td><td><pre>li $a0 i</pre></td><td><code>e1 + e2</code></td><td><pre>   cgen(e1)
    push $a0
    cgen(e2)
    $t1 <= top
    add $a0 $t1 $a0
    pop</pre></td></tr>
<tr><td><code>if e1 = e2 then e3 else e4</code></td><td><pre>   cgen(e1)
    push $a0
    cgen(e2)
    $t1 <= top
    pop
    beq $a0 $t1 true_branch
false_branch:
    cgen(e4)
    j   end_if
true_branch:
    cgen(e3)
end_if:</pre></td><td><code>while e1 = e2 loop e3 pool</code></td><td><pre>predicate:
    cgen(e1)
    push $a0
    cgen(e2)
    $t1 <= top
    pop
    bne $a0 $t1 end_while
    cgen(e3)
    j   predicate
end_while:</pre></td></tr>
<tr><td><code>def f(x1, ..., xn) {e}</code></td><td><pre>f_entry:
    move $fp $sp
    push $ra
    cgen(e)
    $ra <= top
    addiu $sp $sp 4n+8
    lw $fp 0($sp)
    jr $ra</pre></td><td><code>f(e1, ..., en)</code></td><td><pre>   push $fp
    cgen(en)
    push $a0
    ...
    cgen(e1)
    push $a0
    jal   f_entry</pre></td></tr>
<tr><td><code>let x : T <- e1 in e2</code></td><td><pre>   cgen(e1, n)
    push $a0
    cgen(e2, n+1)
    pop</pre></td><td>Temporary var <code>x</code> (whose offset is at <code>ofs</code>)</td><td><pre>lw $a0 -ofs($sp)</pre></td></tr>
</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 $x \leftarrow y + z \ (i)$ 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*:

```
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;
    }
}
```

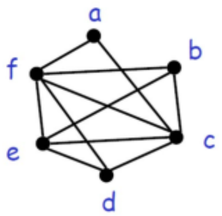
To determine the Liveness of variables through out the Data Flow (i.e. across Basic Blocks), we should apply Dataflow Analysis framework, which will be covered in the last chapter.

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.

- 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 *NP-Hard*. We use the following heuristic algorithm to partially solve this problem:

```
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

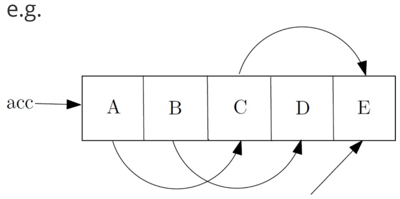
Garbage Collection

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.



- 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:

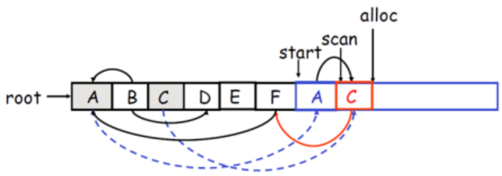
1. Start from Roots, mark all Reachable objects
2. Erase all Unreachable objects, while leaving Reachable ones unmoved

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

Stop & Copy

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

- e.g.



- 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:

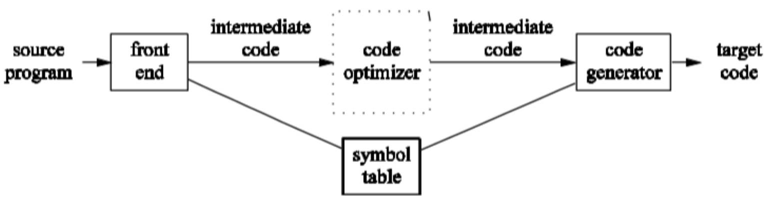
1. A `new` object x has $rc(x) = 1$
2. After each assignment $x \leftarrow y$, $rc(x) - 1$, $rc(y) + 1$
3. When a variable a (pointing to x) goes out out Scope, $rc(x) - 1$
4. Free 0-referenced objects at certain times

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** (过程间优化) 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` ⇒ ~~Deleted~~
    - `x := x * 0` ⇒ `x := 0`
    - `x := x * 2` ⇒ `x := x + x` (Only on machines where `+` is faster than `*`)
    - `x := x ** 2` ⇒ `x := x * x`
    - `x := x * 8` ⇒ `x := x << 3` (Only on machines where `<<` is faster than `*`)
2. **Constant Folding**: compute constant expressions at compile time, e.g.
    - `x := 1 + 2` ⇒ `x := 3`
    - `if 2 < 0 jump Label` ⇒ `if false jump Label` ⇒ ~~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` ⇒ `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` ⇒ `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:

```
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` ⇒ `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:

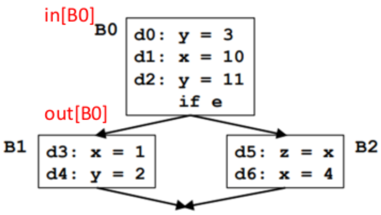
```
void globalOptimization() {
    do {
        do {
            Choose a Local Optimization and perform it;
        } while (still have possible Local Optimizations);
        Choose a Global Optimization and perform it;
    } while (still have possible Optimizations && iteration threshold not met);
}
```

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. $f_B = f_{s2} \circ f_{s1} \circ f_{s0}$
- For each edge $B_0 \to B_e$ 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 $B_0, B_1, \dots, B_n$, then $in[B_e] = out[B_0] \wedge out[B_1] \wedge \dots \wedge out[B_n]$
     - **Meet Operator** $\wedge$ 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 $\wedge$ 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 iff $\exists$ a path $d \to p$ 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".

- in / out: set of Definitions $\{d_0, d_1, \ldots\}$
- $out[s] = f_s(in[s]) = Gen[s] \cup (in[s] - Kill[s])$
  - $Gen[s]$ means the Definition d generated in s (if s is `d: x = ...`)
  - $Kill[s]$ means set of all other Definitions of x in the program
- $\wedge$ is simply Union ($\cup$)

An iterative algorithm can be:

```
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);
}
```

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

Scenery: Liveness Analysis

A Variable `v` is **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 **Dead** and that can trigger Dead Code Elimination. The problem of **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 $in[\text{exit}]$
> - Transfer Function reversed, i.e. $in[B] = f_B(out[B])$
> - Meet Operations occur at startpoints of edges

- in / out: set of Live Variables $\{v_0, v_1, \ldots\}$
- $in[s] = f_s(out[s]) = Use[s] \cup (out[s] - Def[s])$
  - $Use[s]$ means set of all Variables ($\{y, z\}$) used at s (if s is `x = y + z`)
  - $Def[s]$ means the Variable defined at s (x)
- $\wedge$ is simply Union ($\cup$)

An iterative algorithm can be:

```
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);
}
```

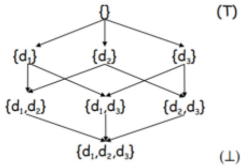
Scenery: "Must-reach" Definitions

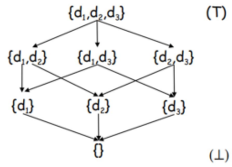
A Definition d "Must-reach" a point p iff $\forall$ paths $\rightarrow p$, d appears at least once and will not be overwritten. In this case:

- $\wedge$ should be $\cap$
- All other setups are the same as Reaching Definitions

Semi-Lattice Diagram

Dataflow Analysis framework can be represented as a mathematical **Meet Semi-Lattice** (最大下界半格) Diagram. That Semi-Lattice is a **Partially-ordered** (偏序的) set which has a **Greatest Lower Bound** (i.e. Meet) for $\forall$ *finite* subset.

Example 1 ($\wedge = \cup$): 

Example 2 ($\wedge = \cap$): 

- **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 $y \to x$
  - If Meet Operation is $\cup$, *largest* subset (i.e. **Top** $\top$) is $\emptyset$, and *smallest* subset (i.e. **Bottom** $\bot$) is the whole Domain
  - If Meet Operation is $\cap$, then *largest* is the whole Domain, and *smallest* is $\emptyset$

Meet Operator follows several properties:

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

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

1. Reflexive (自反): $x \leq x$
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 `Undef` / `NAC` (NOT a Constant) / Constant c
- Transfer Function f is defined as:
  - For non-assignment statement s, f_s is *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 `NAC`, then $v'_x =$ `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 `NAC` and any of them is `Undef`, then $v'_x =$ `Undef`
    - Else (e.g. e is a function call), $v'_x =$ `NAC`
- Meet Operation $v_x \wedge v_y$ is defined as:
  - If any of them is `NAC`, then $v_x \wedge v_y =$ `NAC`
  - If any of them is `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 $v_x \wedge v_y =$ `NAC`
  - If $v_x = v_y = c$, then $v_x \wedge v_y =$ Constant c

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