Review: SUSTech CS323. Compiler: Principles, Techniques and Tools
Author: IskXCr@outlook.com
Image Credit: CS323 Slides/Compiler: Principles, Techniques and Tools

1 Introduction

1.1 Category of Programming Languages:

- *Low-level*. Directly understandable by a computer.
  - **Machine Language**.
  - **Assembly**. Mnemonic names for machine instructions. Macro instructions for frequently-used sequences of machine instructions. Explicit manipulation of memory addresses and content. *Machine dependent*.
- *High-level*. Understandable by human beings. Translator required to be understood by ac computer.
  - **Fortran**: Scientific computation. **Cobol**: Business data processing. **Lisp**: Symbolic Computation.

1.2 Structure of a Compiler:

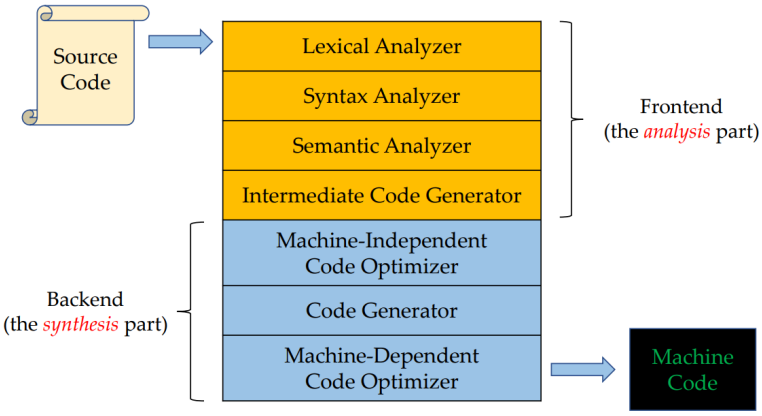
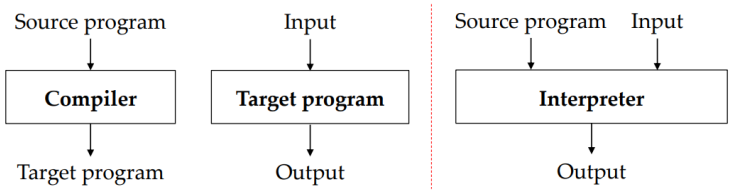


Figure 1: The Structure of a Compiler

- **Frontend**. Breaks up the source program into constituent pieces and imposes a *grammatical structure* on them. Collects *information* about the source program, including but not limited to *list of symbols*. Creates an **intermediate representation (IR)** of the source program.
  - **Lexical Analysis**. Breaks down the source code into a sequence of *lexemes*. Produce a *token* for each lexeme, in the form `<token-name, attributes>`.
  - **Syntax Analysis**. *Check form of the program*. Create an intermediate representation that depicts the grammatical structure of the token stream, typically a *syntax tree*.
  - **Semantic Analysis**. *Check the meaning of the program*.
    - *Type Checking*. Check each operator has matching operands of correct types.
    - *Type Conversion*. Convert types if required.
    - ...
  - **Intermediate Code Generation**. Generate an intermediate representation, typically *three-address code*.
    - *Assembly-like instructions* with three operands per instruction.
    - *Virtual registers*. Each operand acts like a register.
- **Backend**. Constructs the target program from the **IR**. Optimizations.
  - **IR Optimization**. Improve generated IR for better *target code*.

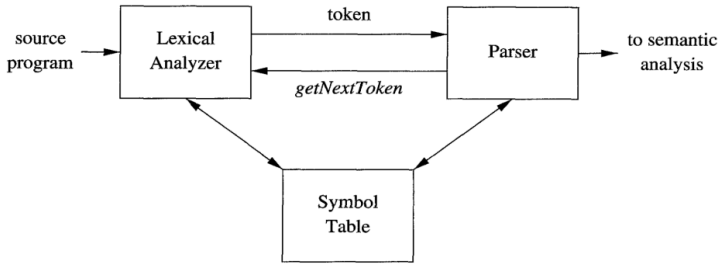
1.3 Compilers vs. Interpreters

- **Code Translation**.
  - *Compiler*. Translate source code directly into *machine code* that can be run directly on target machine.
  - *Interpreter*. Executes each statement in the source code. Source code need not to be necessarily machine code.



- **Code Analysis**.
  - *Compiler*. Typically, analyze the entire program (CFG, ...) to enable optimizations.
  - *Interpreter*. Often takes less time to analyze source code.
- **Semantic/Runtime Error Checking**.
  - *Compiler*. Only executed after passing static analysis.
  - *Interpreter*. Often continue execution until the first met error.

2 Lexical Analysis



Read the input characters of the source program, group them into lexemes, and produces a sequence of tokens. (*Sometimes also register symbols.*)

2.1 Concepts

- **Lexeme**. The lowest syntactic unit, consists of a string of characters.
- **Token**. Syntactical category representing classes of lexemes: `<token name, attribute value>`.
  - *Token name*. An abstract symbol representing the kind of the token. Influence parsing decisions.
  - *Attribute value (optional)*. Containing attributes, e.g., name of an ID token. Influence semantic analysis, code generation, etc.
- **Pattern**. A description of the form that the lexemes of a token may have.

2.1.1 Strings and languages

- **Alphabet**. Any *finite* set of symbols.
  - Example of symbols: letters, digits and punctuations.
  - Example of alphabets: `{1,0}`, ASCII, Unicode
- **String**. A string over an alphabet is a *finite* sequence of symbols drawn from the alphabet.
  - *Length*. For string s, the length is the number of symbols in s and is denoted $|s|$.
  - *Empty String*. String of length 0, ε.
- **Prefix** of s. Any string obtained by removing 0 or more symbols from the end of s.
- **Proper prefix** of s. A prefix that is not ε and not s itself.
- **Suffix**. Similar to prefix, but by removing 0 or more symbols from the beginning of s.
- **Proper suffix** of s. A suffix that is not ε and not s itself.
- **Substring** of s. Any string obtained by removing any prefix and any suffix from s.
- **Proper substring** of s. A substring that is not ε and not s itself.
- **Subsequence**. Any string formed by removing 0 or more not *necessarily consecutive* symbols from s.
- **Concatenation**. The concatenation of string x and y is denoted by xy.
- **Exponentiation**. $s^0 = \varepsilon$, $s^1 = s$, $s^i = s^{i-1}s$ (concatenation of s^{i-1} and s).
- **Language**. A language is any *countable set* of strings over some fixed alphabet.
  - Language with its set $\{\varepsilon\}$, containing only the empty string, is denoted $\emptyset$.

OPERATION	DEFINITION AND NOTATION
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

2.2 Regular Expressions

Rules that define regexps over an alphabet Σ:

- **BASIS**: two rules form the basis:
  - ε is a regexp, $L(\varepsilon) = \{\varepsilon\}$
  - If a is a symbol in Σ, then a is a regexp, and $L(a) = \{a\}$.

- **INDUCTION**: Suppose r and s are regexps denoting the languages $L(r)$ and $L(s)$
  - $(r)|(s)$ is a regexp denoting the language $L(r) \cup L(s)$
  - $(r)(s)$ is a regexp denoting the language $L(r)L(s)$
  - $(r)^*$ is a regexp denoting $(L(r))^*$
  - (r) is a regexp denoting $L(r)$. Additional parentheses do not change the language an expression dentoes.
- **Operator Precedence**. closure $*$ > concatenation > union $\cup$ or $|$.
- **Operator Associativity**. All three operators are *left associative*. Operations are grouped from the left. $a \mid b \mid c$ will be interpreted as $(a \mid b) \mid c$.

2.3 Regular Language

A language that can be defined by a regexp is called a *regular language*. If two regexps r and s denote the same language, then they are *equivalent* and the relationship is written as $r = s$.

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt; (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\varepsilon r = r\varepsilon = r$	ε is the identity for concatenation
$r^* = (r\|\varepsilon)^*$	ε is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

2.4 Regular Definition

A *regular definition* over an alphabet of basic symbols Σ, is a sequence of definitions of the form

$$d_1 \to r_1, d_2 \to r_2, ..., d_n \to r_n,$$

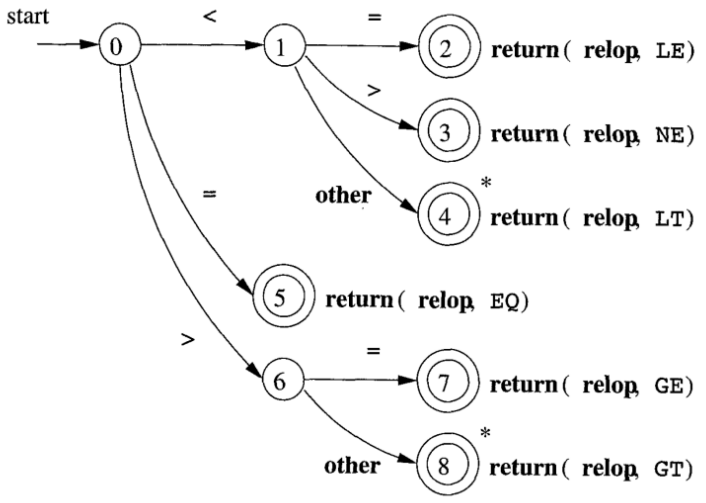
where each d_i is a new symbol not in Σ and not the same as other d's, and each r_i is a regexp over the alphabet $\Sigma \cup \{d_1, ..., d_{i-1}\}$.

Being a sequence implies reusing *previously defined* regular definitions to construct a *new* regular definition.

Notational extensions. *They are only for notational convenience.*

- *One of more instances*. The unary postfix operator $+$. $r^+ = rr^*$.
- *Zero or one instance*. The unary postfix operator $?$. $r? = r \mid \varepsilon$.
- *Character classes*. Shorthand for a logical sequence.
  - $[a_1a_2...a_n] = a_1 \mid a_2 \mid ... \mid a_n$
  - $[a\text{-}e] = a \mid b \mid c \mid ... \mid e$

2.5 Transition Diagrams



Patterns are converted into transition diagrams. Each node corresponds to a **state** of a finite automaton.

- **Start State** (Indicated by an edge labeled *start*), **Accepting State** (Double circles).
- **The Retract Action**. Retract the character pointer to preserve the extra character that should not be consumed by the current lexeme.

2.5.1 Finding keywords

- **Sequential**. Try transition diagram for each type of token sequentially.
  - `fail()` resets the pointer `forward` and tries the next diagram.
  - *Not efficient.*
- **Multiple Parallel Transition**. Run transition diagrams in parallel.
  - Take the longest prefix of the input that macthes any pattern.
  - *Requires special hardware, and may degenerate to the sequential strategy in certain cases.*

2.6 Finite Automata

Finite automata are the simplest machines to recognize patterns.

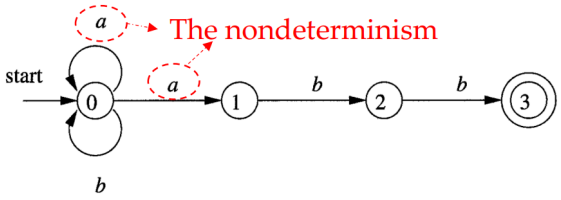
- **Nondeterministic Finite Automata (NFA)**. A symbol can label several edges out of the same state, and ε is a possibel state.
- **Deterministic Finite Automata**. For each state and for each symbol in the input alphabet, there is exactly one edge with that symbol leaving that state.

NFAs and DFAs recognize *regular languages* described by regexps.

2.6.1 NFA

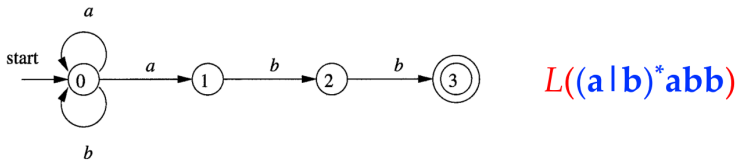
An NFA is a 5-tuple, consisting of

1. A finite set of states S,
2. A set of input symbols Σ, the *input alphabet*. (Assumption: $\varepsilon \notin \Sigma$).
3. A *transition function* that gives a *set* of *next states* for each (s, a) where $s \in S$ and $a \in \Sigma$.
4. A *start state* (initial state) $s_0 \in S$.
5. A set of *accepting states* (final states) $F \subset S$.



An NFA *accepts* an input string x IFF there is a path in the transition graph from the start state to one accepting state, such that the symbols along the path form x (ε labels are ignored).

The *language defined/accepted by an NFA* is the set of strings labelling some paths from the start state to the final states.



2.6.2 DFA

A DFA is a **special NFA** where:

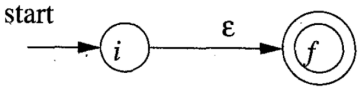
- There are no moves on input ε.
- For each *state* s and input *symbol* a, there is exactly one edge out of s labeled a (exactly **one target state**).

2.7 Converting Regular Expressions to DFAs

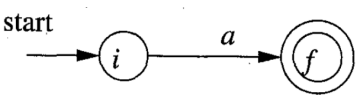
2.7.1 Thompson's Construction Algorithm (RegExp to NFA)

2.7.1.1 Two basis rules

1. The *empty expression* ε is converted to

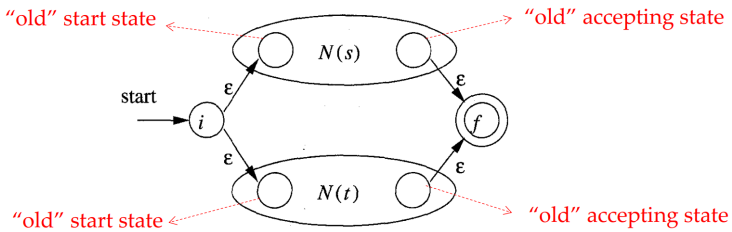


2. Any subexpression a (a *single symbol* in the input alphabet) is converted to



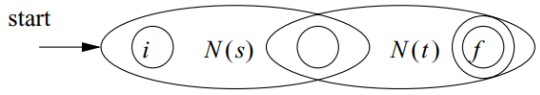
2.7.1.2 The inductive rules

1. **Unions**. $s \mid t$: $N(s)$ and $N(t)$ are NFAs for suexpressions s and t, respectively.



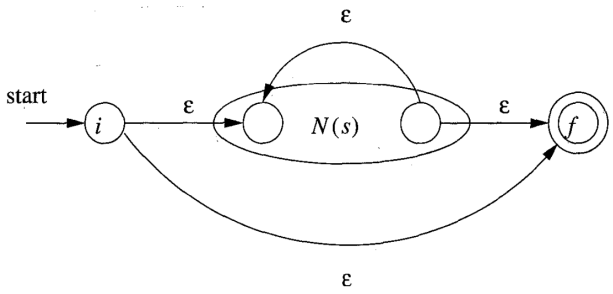
By this construction, the NFAs have only one start state and one accepting state.

2. **Concatenations**. st: $N(s)$ and $N(t)$ are NFAs for subexpressiosn s and t.



Merges the accepting state of $N(s)$ and the start state of $N(t)$.

3. **Kleene Closures**. s^*: $N(s)$ is the NFA for subexpression s.



2.7.2 The Subset Construction Algorithm (NFA to DFA)

This algorithm **simulates "in parallel" all possible** moves an NFA can make on a given input string to map a set of NFA states to a DFA states.

2.7.2.1 Operations

- ε-closure(s): Set of NFA states reachable from NFA state s on ε-transitions alone.
- ε-closure(T): Set of NFA states reachable from some NFA state $s \in T$ on ε-transitions alone. In other words, $\cup_{s \in T}$ ε-closure(s).

```
EPSILON-CLOSURE(T):
1   push all states of T onto stack;
2   initialize ε-closure(T) to T;
3   while (stack is not empty)
4       pop t, the top element, off stack;
5       for ( each state u with an edge from t to u labeled ε)
6           if (u is not in ε-closure(T))
7               add u to ε-closure(T);
8               push u onto stack;
9       end
10  end
```

It's just DFS.

- move(T, a): Set of NFA states to which there is a transition on input symbol a from some states $s \in T$, i.e., the target states of those states in T when seeing a.

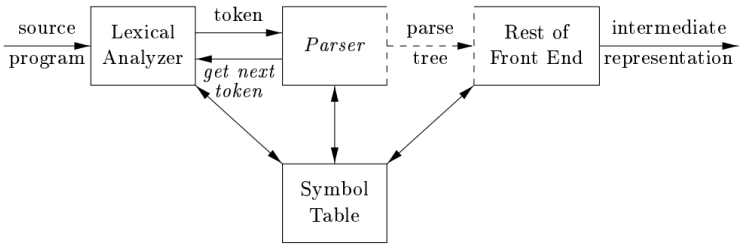
2.7.2.2 The Algorithm

An accepting state of the constructed DFA corresponds to a subset of the NFA states, in which there exists at least one accepting NFA state.

- If there are more than one accepting NFA state, then there exists an *conflict*, in which theres exists an input string, such that the prefix of which matches multiple patterns.
- *Upon conlifcts, find the first pattern whose accepting state is in the set and make that pattern the output of the DFA state.*

```
SUBSET-CONSTRUCTION():
1   Dstates ← {ε-closure(s0)};
2   while (there is an unmarked state T in Dstates)
3       mark T;
4       for ( each input symbol a)
5           U = ε-closure(move(T, a));
6           if (U is not in Dstates)
7               add U as an unmarked state to Dstates;
8           Dtran[T, a] = U;
9       end
10  end
```

3 Syntax Analysis



The parser obtains a string of tokens from the lexical analyzer and verifies that the srting of token names can be generated by the grammar for the source language.

- **Universal Parsers**.
  - Some methods (e.g., Earley's algorithm) can parse any grammar. However, they are too inefficient to use in practice.
- **Top-Down Parsers**.
  - Construct parse trees from the top (root) to the bottom (leaves).
- **Bottom-Up Parsers**.
  - Construct parse trees from the bottom (leaves) to the top (root).

3.1 Context-Free Grammar (CFG)

3.1.1 Structure of CFGs

A CFG consists of four parts:

- **Terminals**. Basic symbols from which strings are formed. *Token names.*
- **Nonterminals**. Syntactic variabels that denotes sets of strings. *Usually correspond to a language construct, such as* `stmt` *(statements).*
- **Start Symbol**. One nonterminal is distinguished as the *start symbol.*
  - The set of strings denoted by the start symbol is the language generated by the CFG.
- **Productions**. Specify how the terminals and nonterminals can be combined to form strings.
  - **Format**: head $\rightarrow$ body
  - **head** must be a *nonterminal.* **body** consists of zero or more terminals/nonterminals.
  - *Example.* expression $\rightarrow$ expression + term.
- **Operators**.
  - Alternative specification: |

3.1.2 Derivation and Parse Tree

- **Notation**.
  - $\Rightarrow$: derives in *one* step
  - $\overset{*}{\Rightarrow}$: derives in ***zero*** *or more* steps.
  - $\overset{+}{\Rightarrow}$: derives in ***one*** *or more* steps.
- **Derivation**. Starting with the start symbol, nonterminals are rewritten using productons until only *terminals remain.*
  - *Example.* CFG $E \rightarrow -E \mid E + E \mid E * E \mid (E) \mid \text{id}$
  - *Derivation.* $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\text{id})$

3.1.3 Sentential Form and Sentence

Sentential Form: If $S \overset{*}{\Rightarrow} \alpha$, where S is the start symbol of a grammar G, then α is *sentential form* of G.

- Sentential form may contain both terminals and nonterminals, and may be empty.
- *Example.* $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathrm{id}+E) \Rightarrow -(\mathrm{id}+\mathrm{id})$. All strings here are sentential forms.

Sentence: A *sentence* of G is a sentential form with no nonterminals.
- *Example.* $-(\mathrm{id}+\mathrm{id})$

Language of a Grammar: The *language* generated by a grammar is its set of sentences.

3.1.4 Leftmost/Rightmost Derivations

Leftmost Derivation. The leftmost nonterminal in each sentential form is always chosen to be replaced.
- $E \underset{\text{lm}}{\Rightarrow} -E \underset{\text{lm}}{\Rightarrow} -(E) \underset{\text{lm}}{\Rightarrow} -(E+E) \underset{\text{lm}}{\Rightarrow} -(\mathrm{id}+E) \underset{\text{lm}}{\Rightarrow} -(\mathrm{id}+\mathrm{id})$

Rightmost Derivation. The rightmost nonterminal is always chosen to be replaced.
- $E \underset{\text{lm}}{\Rightarrow} -E \underset{\text{lm}}{\Rightarrow} -(E) \underset{\text{lm}}{\Rightarrow} -(E+E) \underset{\text{lm}}{\Rightarrow} -(E+\mathrm{id}) \underset{\text{lm}}{\Rightarrow} -(\mathrm{id}+\mathrm{id})$

3.1.5 Parse Tree

A *parse tree* is a graphical representation of a derivation that filters out the order in which productions are applied.
- **Root Node**. The start symbol of the grammar.
- **Leaf Node**. Labeled by a terminal symbol.
- **Interior Node**. Labeled with a nonterminal symbol and represents the application of a production. (Children of this node are labeled, from left to right, the body of the production)

The leaves, from left to right, constitute a **sentential form** of the grammar, which is called the *yield* or *frontier* of the tree.
- There is a **many-to-one** relationship between derivations and **parse trees**.
  - However, there is a **one-to-one** relationship between *leftmost/rightmost derivations* and *parse trees*.

3.1.6 Ambiguity

Given a grammar, if there are *more than one* parse tree for some sentence, then it is ambiguous.
- Example CFG: $E \to E+E \mid E*E \mid (E) \mid \mathrm{id}$
- Eliminating

3.1.7 CFG vs RegExp

- CFGs are **more expressive** than regular expressions.
  1. Every language that can be described by a regular expression can also be described by a CFG.
  2. Some context-free languages (CFLs), cannot be described using regular expressions.
     - *Example.* $L = \{a^n b^n \mid n > 0\}$.
     - *Proof by contradiction.* Suppose there are k (finite) states. Try parsing a string s of length $|s| > k$.

3.1.8 Grammar Design

3.1.8.1 Eliminating Ambiguity

- **Principle of Promixity**. Match `else` with the closest unmatched `then`.

3.1.8.2 Eliminating Left Recursion

Grammar with left recursion cannot be handled by top-down parsers.
- **Left Recursive**. A grammar is left recursive if it has a nonterminal A such that there exists a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α
  - $S \to Aa \mid b$
  - $A \to Ac \mid Sd \mid \varepsilon$
- **Immediate Left Recursion**. A production of the form $A \to A\alpha$.
- **General Left Recursion Elimination**.

  $$A \to A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$$

  Replaced by

  $$A \to \beta_1 A' \mid \ldots \mid \beta_n A'$$
  $$A' \to \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

- **Left Factoring**.
  If $A \to \alpha\beta_1 \mid \alpha\beta_2$ are two productions, and the input begins with a nonempty string derived from α. We may defer choosing productions by expanding A to $\alpha A'$ first.

  $$A \to \alpha A' \qquad A' \to \beta_1 \mid \beta_2$$

3.2 Top-Down Parsing

Construct a parse tree for the input string, **starting from the root** and creating nodes of the parse tree in *preorder* (depth-first).

Two **basic actions** of top-down parsing algorithms:
- **Predict**. At each step of parsing, determine the production to be applied for the *leftmost nonterminal* of the parse tree's frontier.
- **Match**. Match the terminals in the chosen production's body with the input string.

3.2.1 Recursive-Descent Parsing, RDP

Procedures. RDP algorithms has a set of *procedures*, one for each nonterminal that deals with a substring of the input.

Execution. Execution begins with the procedure for the start symbol.
- **Announce success** if the procedure scans the entire input (the start symbol derives the whole input via applying a series of prodution).

```
RECURSIVE-RANDOM-DESCENT-PARSING(A):
1  Choose an A-production, A → X1X2···Xk;
2  for (i = 1 to k)
3      if (Xi is a nonterminal)
4          call procedure Xi()
5      else if (Xi equals the current input symbol a)
6          advance the input to the next symbol;
7      else /* An error occurred */;
8  end
```

3.2.2 Computing $\mathrm{FIRST}(X)$

3.2.2.1 Computing for Grammar Symbols

$\mathrm{FIRST}(X)$, where X is a grammar symbol:
- If X is a **terminal**, then $\mathrm{FIRST}(X) = \{X\}$
- If X is a **nonterminal** and $X \to \varepsilon$, then add ε to $\mathrm{FIRST}(X)$
- If X is a **nonterminal** and $X \to Y_1 Y_2 \cdots Y_k (k \geq 1)$ is a production:
  - If for some i, a is in $\mathrm{FIRST}(Y_i)$ and ε is in all of $\mathrm{FIRST}(Y_1), \cdots, \mathrm{FIRST}(Y_{i-1})$, then add a to $\mathrm{FIRST}(X)$.
  - If ε is in all of $\mathrm{FIRST}(Y_1), \cdots, \mathrm{FIRST}(Y_k)$, then add ε to $\mathrm{FIRST}(X)$

3.2.2.2 Computing for Concatenations

$\mathrm{FIRST}(X_1 X_2 \cdots X_n)$, where $X_1 X_2 \cdots X_n$ is a string of grammar symbols:
- Add all non-ε symbols of $\mathrm{FIRST}(X_1)$ to $\mathrm{FIRST}(X_1 X_2 \cdots X_n)$
- If ε is in $\mathrm{FIRST}(X_1)$, add non-ε symbols of $\mathrm{FIRST}(X_2)$ to $\mathrm{FIRST}(X_1 X_2 \cdots X_n)$
- If ε is in $\mathrm{FIRST}(X_1)$ and $\mathrm{FIRST}(X_2)$, add non-ε symbols of $\mathrm{FIRST}(X_3)$ to $\mathrm{FIRST}(X_1 X_2 \cdots X_n)$.
- ...
- If ε is in $\mathrm{FIRST}(X_i)$ for all i, add ε to $\mathrm{FIRST}(X_1 X_2 \cdots X_n)$

3.2.3 Computing $\mathrm{FOLLOW}(X)$

Computing FOLLOW set for all nonterminals:
- Add \$ in $\mathrm{FOLLOW}(S)$, where S is the start symbol and \$ is the input **right endmarker**.
- Apply the rules below, until all FOLLOW sets do not change:
  1. If there is a production $A \to \alpha B \beta$, then everything in $\mathrm{FIRST}(\beta)$ except ε is in $\mathrm{FOLLOW}(B)$
  2. If there is a production $A \to \alpha B$ (or $A \to \alpha B \beta$ and $\mathrm{FIRST}(\beta)$ contains ε), then everything in $\mathrm{FOLLOW}(A)$ is in $\mathrm{FOLLOW}(B)$

3.2.4 LL(1) Grammars

Recursive-descent parsers needing no backtracking can be constructed for a class of grammars called **LL(1)**.
- Left-to-right, Leftmost Derivation, **1** input symbol of **lookahead**.

A grammar G is LL(1) IFF **for any two distinct productions** $A \to \alpha \mid \beta$, the following conditions hold:
- There is no terminal a such that α and β derive strings beginning with a
- At most one of α and β can derive the empty string
- If $\beta \overset{*}{\Rightarrow} \varepsilon$,then α does not derive any string beginning with a terminal in $\mathrm{FOLLOW}(A)$ and vice versa.

3.2.5 Parsing Table

We may build parsing tables for recursive-descent parsers.

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

For an LL(1) parser, there are **no entries with multiple productions**.

CONSTRUCT-PARSING-TABLE(G):

```
1  for (each production A → α of G)
2      for (each terminal a in FIRST(α))
3          add A → α to M[A, a];
4      end
5      if ε is in FIRST(α)
6          for (each terminal b ∈ FOLLOW(A) (including $))
7              add A → α to M[A, b];
8          end
9  end
10 Set all empty entries in the table to error;
11 return M;
```

- **Conflicts**. If there arise a conflict during the above construction, then the input grammar is **not** LL(1).

3.2.6 Non-Recursive Predictive Parsing

We may explicitly maintain a **stack** to build a non-recursive predictive parser.

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Figure 17: Matched content + symbols in stack always forms a *left-sentential form* in a non-recursive predictive parser. **The top of the stack is to the left.**

NON-RECURSIVE-PREDICTIVE-PARSING(ω):

```
1  let a ← first symbol of ω;
2  let X ← top stack symbol;
3  while (X ≠ $)
4      if (X = a)
5          pop the stack and let a be the next symbol of ω;
6      else if (X is a terminal)
7          error();
8      else if (M[X, a] is an error entry)
9          error();
10     else if (M[X, a] = X → Y1Y2···Yk)
11         output the production X → Y1Y2···Yk;
12         pop the stack;
13         push Yk, Yk−1, ..., Y1 onto the stack, with Y1 on top;
14     let X be the top stack symbol;
15 end
```

3.3 Bottom-Up Parsing

Construct a parse tree for an input string beginning at the leaves (**terminals**) and working up towards the root (**start symbol of the grammar**).

3.3.1 Shifting-Reduce Parsing, SRP

A *general style* of bottom-up parsing (using a *stack* to hold grammar symbols).

Two **basic actions**:

- **Shift**. Move an input symbol onto the stack.
- **Reduce**. Replace a string at the stack top with a non-terminal that can produce the string. (The reverse of a rewrite step in a derivation).

Characteristics:

- A **stack** holds grammar symboles
- An **input buffer** holds the rest of the string to be parsed
- The *stack content* (from bottom to top) and the *input buffer content* form a **right-sentential form** (assuming no errors).

3.3.2 LR(k) Parsers

LR(k) parsers: the most prevalent type of bottom-up parsers:

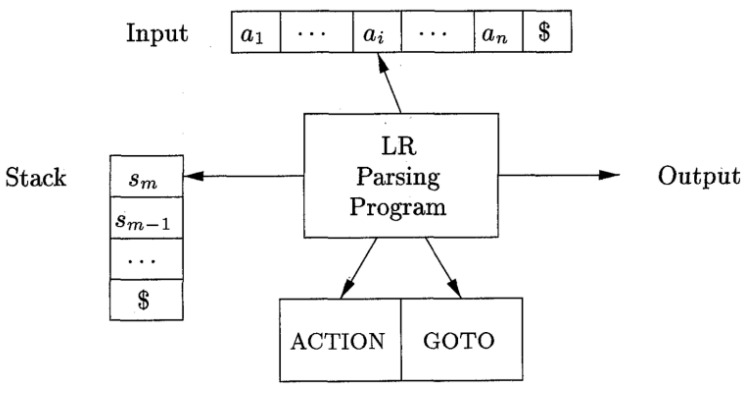
- **L**. Left-to-right scan of the input
- **R**. Construct a rightmost derivation in reverse
- k. Uses k input symbols of lookahead in making parsing decisions.

Characteristics:

- Table-driven & powerful
- Nonbacktracking
- Recognize *virtually all* programming language constructs for which CFGs can be written
- Describes *more* languages than LL grammars.

An LR parser makes shift-reduce decisions by **maintaining states** to keep track of **what have been seen** during parsing.

- The states are **sets of LR(k) items.**



An LR parser consists of:

- An **input**, an **output**, a **stack**, a **driver program**, and a **parsing table**.
- The driver program is the same for all LR parsers. Only parsing table changes.
- The stack holds a sequence of states.

- The parser decides the next action based on:
  1. The state at the top of the stack, and
  2. The **next k** terminal read from the input buffer.

Configuration. The *configuration* of a LR parser represents the complete state of the parser, i.e., **stack status + input status**. A configuration is a pair:

Stack contents (top on the right) $(s_0 s_1 \dots s_m, a_i a_{i+1} \dots a_n \$)$ Remaining input

Each state except s_0 in an LR parser corresponds to a set of items and *a grammar symbol* (the symbol that leads to the state transition).

- Suppose X_i is the grammar symbol for state s_i, then $X_0 X_1 \cdots X_m a_i a_{i+1} \cdots a_n$ is a **right-sentential form** (assume no errors).

3.3.2.1 Augmented Grammar

The augmented version of grammar G with start symbol S is given by:

- Introduce a **new start symbol** S' to take the role of S, and
- Add a new production $S' \to S$.

3.3.2.2 LR(0) Items

An LR(0) item is a production with a dot at some position of the body, indicating how much we have seen at a given time point in the parsing process.

- $A \to \cdot XYZ$ $\quad A \to X \cdot YZ$ $\quad A \to XY \cdot Z$ $\quad A \to XYZ \cdot$
- $A \to X \cdot YZ$: we have just seen on the input a string derivable from X, and we hope to see YZ next.

3.3.2.3 Canonical LR(0) Collection

The collection of states (sets of LR(0) items) is called the *canonical* LR(0) collection, and it provides the basis for constructing a DFA to make parsing decisions.

To construct canonical LR(0) collection for a grammar, we need to define

- An augmented grammar
- Two functions: (1) CLOSURE of items sets and (2) GOTO

Closure of item sets. If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by two rules:

1. Initially, add every item I to CLOSURE(I)
2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot \gamma$ to CLOSURE(I), if it is not already there. *Apply this rule until no more new items can be added to* CLOSURE(I).

```
CLOSURE(I):
1  J ← I;
2  repeat
3      for (each item A → α · Bβ in J)
4          for (each production B → γ of G')
5              if (B → · γ is not in J)
6                  add B → · γ to J;
7  until no more items are added to J on one round
8  return J;
```

The Function GOTO. GOTO(I, X) is defined to be the closure of the set of all items $[A \to \alpha X \cdot \beta]$ where $[A \to \alpha \cdot X\beta]$ is in I.

- CLOSURE$(\{[A \to \alpha X \cdot \beta] \mid [A \to \alpha \cdot X\beta] \in I\})$

Constructing Canonical LR(0) Collection.

```
CONSTRUCT-CANONICAL-LR(0)-ITEM-SET(G'):
1  C ← {CLOSURE({[S' → · S]})};
2  repeat
3      for (each set of items I ∈ C)
4          for (each grammar symbol X)
5              if (GOTO(I, X) is not empty and not in C)
6                  add GOTO(I, X) to C;
7  until no new sets of items are added to C on a round.
```

3.4 Simple LR, SLR

Construct the LR(0) automaton from the grammar G.

- **States**. The item sets in the canonical LR(0) collection.
- **Transitions**. Given by the GOTO function.
- **Start State**. CLOSURE$(\{[S' \to \cdot S]\})$
- **Parsing Table**. ACTION + GOTO
  - ACTION(i, a). Can have one of the four types of values:
    - **Shift** j. Shift input a to the stack, and uses state j to represent a.
    - **Reduce** $A \to \beta$. Reduce β on top of the stack to non-terminal A.
    - **Accept**. The parser accepts the input and finishes parsing.
    - **Error**. Syntax errors exist.
  - GOTO(i, a).
    - If GOTO$(I_i, a) = I_j$, then GOTO$(i, a) = j$.

For the configuration, the SLR parser checks ACTION$[s_m, a_i]$ in the parsing table to decide the parsing action.

- **Shift** s. *Shift* the next state s onto the stack, entering the configuration
$$(s_0 s_1 \cdots s_m s, a_{i+1} \cdots a_n)$$
- **Reduce** $\alpha \to \beta$. Execute a *reduce* move, entering the configuration
$$(s_0 s_1 \cdots s_{m-r} s, a_i a_{i+1} \cdots a_n),$$
where $r = |\beta|$ and $s =$ GOTO(s_{m-r}, A).
- **Accept**. Parsing succeeded.
- **Error**. Found an error *and calls an error recovery routine.*

Input: The parsing table for a grammar G and an input string ω.
Output: If ω is in $L(G)$, the reduction steps of a bottom-up parse for ω; otherwise, an error indication.
Initial Configuration. $(s_0, \omega\$)$

```
LR-PARSING-ALGORITHM(G, ω):
1  while (1)
2      let s ← state on top of the stack;
3      if (ACTION(s, a) = shift t)
4          push t onto the stack;
5          let a be the next input symbol;
6      else if (ACTION(s, a) = reduce A → β)
7          pop |β| symbols off the stack;
8          let state t now be on top of the stack;
9          push GOTO[t, A] onto the stack;
10         output the production A → β;
11     else if (ACTION[s, a] = accept)
12         break;
13     else
14         call error-recovery routine;
15 end
```

3.4.1 Constructing a SLR(1) Parsing Table

The SLR-parsing table for a grammar G can be constructed based on the LR(0) item sets and LR(0) automaton.

1. Construct the canonical LR(0) collection $\{I_0, ..., I_n\}$ for the augmented gramamr G'.
2. State i is constructed from I_i. ACTION can be determined as follows:
   - If $[A \to \alpha \cdot a\beta]$ is in I_i and GOTO$[I_i, a] = I_j$, then set ACTION$[i, a] \leftarrow$ shift j. a **must be a terminal**.
   - If $[A \to \alpha \cdot]$ is in I_i, then set ACTION$[i, a] \leftarrow$ reduce $A \to \alpha$ for **all** a **in** FOLLOW(A).
   - If $[S' \to S \cdot]$ is in I_i, then set ACTION$[i, \$] \leftarrow$ accept.
3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO$(I_i, A) = I_j$, then GOTO$(i, A) = j$.
4. All entries not defined in steps 2 and 3 are set to "error".
5. Initial state is the one constructed from the item set containing $[S' \to \cdot S]$.

If there is no conflict during the parsing table construction, then the grammar is **SLR(1)**.

Weakness of the SLR method. In SLR, the state i calls for reduction by $A \to \alpha$ if

1. the item set I_i contains item $[A \to \alpha \cdot]$ and
2. input symbol a is in FOLLOW(A)

In some situations, after reduction, *the content $\beta\alpha$ on stack top would become βA that cannot be followed by a in any right-sentential form.* (Only requiring "a is in FOLLOW(A)" is **not enough**, if βA cannot be followed by a)

3.5 Canonical LR

3.5.1 LR(1) Items

Carry more information in the state to rule out some invalid reductions (**splitting LR(0) states**)

- General form of an LR(1) item: $[A \to \alpha \cdot \beta, a]$
  - $A \to \alpha\beta$ is a production and a is a terminal or \$.
  - "**1**" refers to the length of the second component: the *lookahead*.
  - The lookahead symbol has no effect if β is not ε, since it only helps determine whetehr to reduce. (a will be inherited during state transitions)
  - An item of the form $[A \to \alpha \cdot, a]$ calls for a reduction by $A \to \alpha$ **only if the next input symbol is** a (the set of such a's is a **subset of** FOLLOW(A))

```
CLOSURE(I):
1  repeat
2      for (each item [A → α · Bβ, a] in J)
3          for (each production B → γ of G′)
4              for ( each terminal b ∈ FIRST(βa))
5                  add B → · γ to set I;
6  until no more items are added to I
7  return I;
```

It only generates the new item $[B \to \cdot \gamma, b]$ from $[A \to \alpha \cdot B\beta, a]$ if $b \in \text{FIRST}(\beta a)$.

```
GOTO(I, X):
1  Initialize J to be the empty set;
2  for (each item [A → α · Xβ, a] ∈ I)
3      add item [A → αX · β, a] to set J;
4  return CLOSURE(J);
```

The lookaead symbols are passed to new items from existing items.

```
CONSTRUCT-LR(1)-ITEM-SET(G′):
1  C ← {CLOSURE({[S′ → · S, $]})};
2  repeat
3      for (each set of items I ∈ C)
4          for (each grammar symbol X)
5              if (GOTO(I, X) ≠ ∅ and GOTO(I, X) ∉ C)
6                  add GOTO(I, X) to C;
7  until no new sets of items are added to C;
```

3.5.2 Constructing a Canonical LR(1) Parsing Table

1. Construct $C' = \{I_0, ..., I_n\}$, the collection of LR(1) item sets for the augmented grammar G'.
2. State i of the parser is constructed from I. Its parsing action is determined as follows:
   - If $[A \to \alpha \cdot a\beta, b]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a] \leftarrow$ shift j.
   - If $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set $\text{ACTION}[i, a] \leftarrow$ reduce $A \to \alpha$.
   - If $[S' \to S \cdot, \$]$ is in I_i, then set $\text{ACTION}[i, \$] \leftarrow$ accept
3. The goto transitions for state i are constructed from all nonterminals A using the rules: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}(i, A) = j$.
4. All entries not defined in steps 2 and 3 are made **error**.
5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S, \$]$.

If any conflict result from the above rules, we say the grammar is **not LR(1)**.

3.6 Lookahead LR (LALR)

The huge set of states of CLR(1)'s parsing table should be simplified.

- Look for sets of LR(1) items with the same ***core***.
  - The *core* of an LR(1) item set is the set of the first components (without the lookahead symbol).
  - *Example.* The core of $\{[C \to c \cdot C, c/d]\}$ is $\{[C \to c \cdot C]\}$.

3.6.1 Merging States in an LR(1) parsing table

Since the core of $\text{GOTO}(I, X)$ depends only on the core of I, the goto targets of merged sets also have the same core and hence can be merged.

3.6.2 Merge Conflicts

Merging states in an LR(1) parsing table may cause conflicts

Merging does not introduce shift/reduce conflicts. Otherwise, the original grammar is not LR(1).

Merging may introduce reduce/reduce conflicts.

3.6.3 Constructing a LALR(1) Parsing Table

1. Construct $C' = \{I_0, ..., I_n\}$, the collection of LR(1) item sets for the augmented grammar G'.
2. For each core present among a set of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let $C' = \{J_0, ..., J_m\}$ be the resulting collection after merging.
   - The parsing actions for state i are constructed from J_i following the LR(1) parsing table construction algorithm.
   - **If there is a conflict, this algorithm fails to produce a parser and the grammar is not LALR(1).**
4. Construct the GOTO table as follows:
   - If J is the union of one or more sets of LR(1), that is $J = I_1 \cup \cdots I_k$, then the cores of $\text{GOTO}(I_1, X), \text{GOTO}(I_2, X), ... \text{GOTO}(I_k, X)$ are the same, since $I_1, ..., I_k$ all have the same core.
   - Let K be the union of all sets of items having the same core as $\text{GOTO}(I_1, X)$
   - $\text{GOTO}(J, X) = K$

3.7 Comparisons Among LR Parsers

The languages (grammars) that can be handled:

$$\text{CLR} > \text{LALR} > \text{SLR}$$

Number of states in the parsing table:

$$\text{CLR} > \text{LALR} = \text{SLR}$$

Driver prorgams:

$$\text{SLR} = \text{CLR} = \text{LALR}$$

4 Syntax-Directed Translation

Syntax-directed translation is the process of language translation guided by context-free grammars.

- **Language translation is in a broader sense**.
- The **semantics** of a construct can be **synthesized** from its constituent cosntructs' semantics.
  - The type of the expression `x + y` is determined by the type of `x` and `y`, and the operator `+`.
- or **inherited** from other constructs (e.g., sibilings in the parse tree)
  - In `int x`, the type of `x` is determined by the type specifier to the left of `x`.

4.1 Syntax-Directed Definitions (SDD)

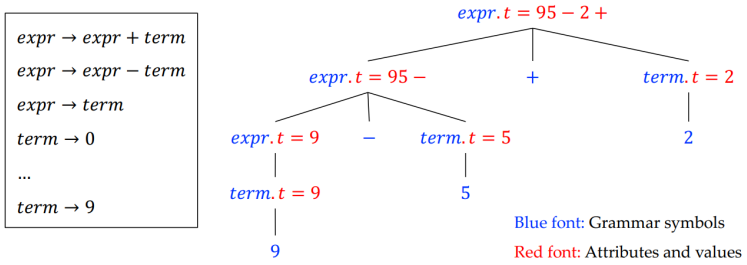
	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (\ E\ )$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

Figure 27: Example of a SDD

A SDD is a context-free grammar together with **attributes** and **rules**.

- **Attributes**. A set of *attributes* is associated with each grammar symbol.
- **Semantic Rule**. A *semantic rule* is associated with a production and describes how attributes are computed.

4.1.1 Annotated Parse Tree



4.1.2 Synthesized Attributes

Synthesized Attributes. The value of such attributes at a tree node N is determined from **the children of** N and at N itself. *Associated with the production of* N.

- **Evaluation Order is Not Specified**. Any order that computes an attribute a after all its dependencies has been computed is fine.
- Value can be evaluated during **a single bottom-up traversal**.

4.1.3 Inherited Attributes

Inherited Attributes. The value of such attributes at a parse-tree node is determined from attribute values at **the node itself**, **its parent** and **its siblings** in the parse tree.

4.2 Evaluation Order of SDDs

In order to compute the attribute a of N, given by $N.a = f(M_1.a_1, ..., M_k.a_k)$, where M_i are other nodes inside the parse tree, we must first compute its dependencies $M_i.a_i$.

4.2.1 Dependency Graphs

- Depict the **information flow** among the attribute instances in a particualr parse tree.
- Model the **partial order** among attribute instances.
- Can be computed by **topological sort**.

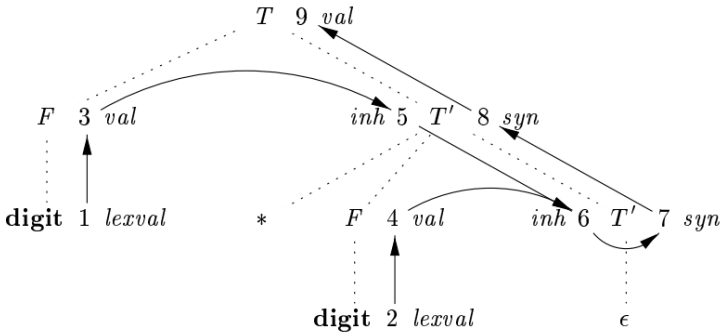


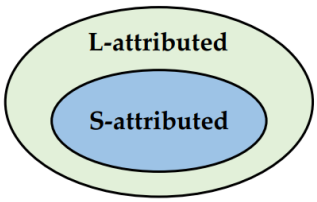
Figure 29: An edge from E_1 to E means that the value of E depends on E_1

4.2.2 S-Attributed SDDs

An SDD is **S-attributed** if every attribute is synthesized.

- The value of such attributes can be computed in any bottom up order in the parse tree, e.g., postorder traversal.
  - Can be done during bottom-up parsing.

4.2.3 L-Attributed SDDs



An SDD is **L-attributed** if each of its attribute is either:

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \to X_1 \cdots X_n$, and that there is an inherited attributed $X_i.a$ computed by a rule associated with this production. Then the rule may use only:
   - Inherited attributes associated with the head A.
   - Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, ..., X_{i-1}$ located to the **left** or X_i.
   - Inherited or synthesized attributes associated with this occurrence of X_i ifself, but only in such a way that there are **no cycles** in a dependency graph formed by the attributes of this X_i.

EVALUATE-L-ATTRIBUTED-SDD(T, n):

```
1  for every child m of n from left to right
2      evaluate the inherited attributes of m
3      EVALUATE-L-ATTRIBUTED-SDD(T, m);   // m's synthesized
4                                         // value will be
5                                         // evaluated
6  end
7  evaluate the synthesized attributes of n;
```

4.3 Syntax-Directed Translation Schemes (SDT)

SDT specify more details on how to do the translation.

An SDT is a context-free grammar with semantic actions (program fragments) embedded within production bodies.

- Differ from the semantic rules in SDD's.

	PRODUCTION		ACTIONS
1)	$S \to$	B	$\{\ B.ps = 10;\ \}$
2)	$B \to$		$\{\ B_1.ps = B.ps;\ \}$
		B_1	$\{\ B_2.ps = B.ps;\ \}$
		B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht);$
			$\quad B.dp = \max(B_1.dp, B_2.dp);\ \}$
3)	$B \to$		$\{\ B_1.ps = B.ps;\ \}$
		B_1 **sub**	$\{\ B_2.ps = 0.7 \times B.ps;\ \}$
		B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$
			$\quad B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps);\ \}$
4)	$B \to$	(	$\{\ B_1.ps = B.ps;\ \}$
		B_1)	$\{\ B.ht = B_1.ht;$
			$\quad B.dp = B_1.dp;\ \}$
5)	$B \to$	**text**	$\{\ B.ht = getHt(B.ps, \mathbf{text}.lexval);$
			$\quad B.dp = getDp(B.ps, \mathbf{text}.lexval);\ \}$

Figure 32: SDT for typesettings boxes. Note that there exists semantic actions after some symbol has been matched and reduced. The major difference is that, the **specific order** of semantic actions and shift/reduce actions are now **specified.**

- Semantic actions are treated as **virtual** parse-tree nodes.
- The evaluation can be done by performing preorder traversal. **However**, in reality it is often done during parsing.
- Reduce conflicts.

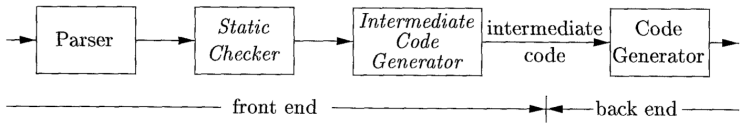
$$
\begin{array}{lll}
L & \to & E\ \mathbf{n} \\
E & \to & \{\ \text{print}('+');\ \}\ E_1 + T \\
E & \to & T \\
T & \to & \{\ \text{print}('*');\ \}\ T_1 * F \\
T & \to & F \\
F & \to & (\ E\ ) \\
F & \to & \mathbf{digit}\ \{\ \text{print}(\mathbf{digit}.lexval);\ \}
\end{array}
$$

Figure 33: Here E and T has reduce conflicts.

Lab contents have been omitted.

5 Intermediate-Code Generation

5.1 Overview of Intermediate Representation (IR)

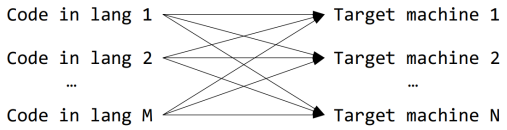


The frontend of a compiler **analyzes a source program** and **creates an intermediate representation**, from which the backend generates target code.

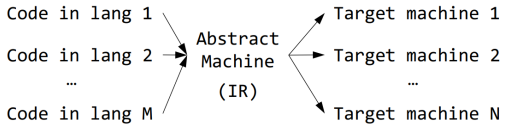
- Details of the source language are confined to the frontend.

- Details of the target machine confined to the backend.

5.1.1 The Benefits of A Common IR

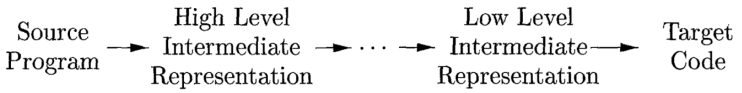


`M * N` compilers without a common IR



`M + N` compilers with a common IR

5.1.2 Different Levels of IRs



A compiler may construct a sequence of IRs.

- **High-Level IRs**. Like syntax trees are close to the source language. Suitable for machine-independent tasks like static type checking.
- **Low-Level IRs**. Close to target machines. Suitable for machine-dependent tasks like *register allocation* and *instruction selection*.

C is often used as an intermediate form.

5.2 DAGs for Expressions

The tree for a common subexpression would be replicated as many times as the subexpression appears. Thus they can be eliminated.

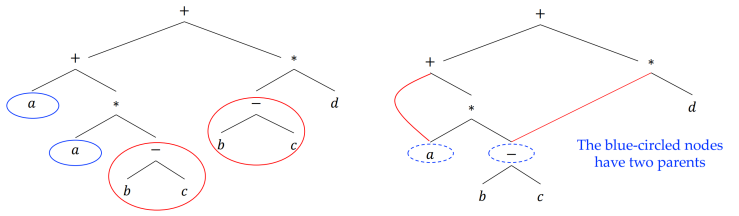


Table 1: Eliminating common subexpressions

5.2.1 Constructing DAG's

A new node is created IFF there is no existing identical node.

5.3 Three-Address Code

In three-address code, there is **at most one operator** on the right side of an instruction.

- Instructions are often in the form $x = y$ op z

Operands (or addresses can be)

- **Names** in the source programs
- **Constants** in the source programs, e.g., numerical constants, string literals, etc.
- **Temporary names** generated by a compiler.

5.3.1 L-values and R-values

- **L-value (location)** refers to the mmory location, which identifies an object.
- **R-value (content)** refers to data value stored at some address in memory.

5.3.2 Quadruples

A *quadruple* has four fields.

- General form: `op arg1 arg2 result`
- `op` contains an **internal code** for the operator
- `arg1, arg2, result` are **addresses** (operands)
- *Example.* $x = y + z$ becomes `+ y z x`.

Exceptions:

- *Unary operators* like $x = \pm y$ do not use `arg2`.
- `param` *operators*, which specifies function arguments, do not use `arg2` or `result`.
- *Conditional/unconditional jumps* put the target label in `result`.

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

5.3.3 Triples

A *triple* has only three fields: `op, arg1, arg2`

- Refer to the result of an operation `x op y` by its position without generating temporary names (an *optimization* over quadruples).

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Note that ***positions needs to be updated if reordering occurs****.*

5.3.4 Indirect Triples

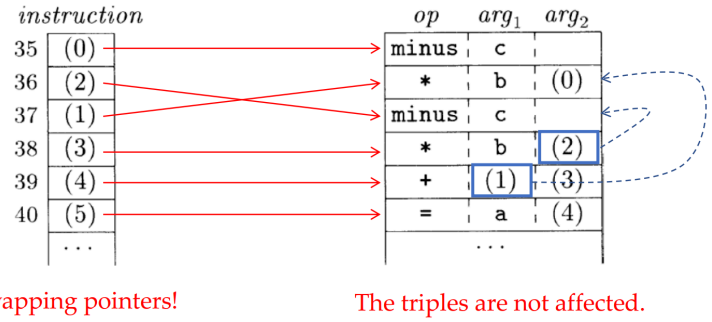


Figure 39: Managing a *instruction list* allows reordering during optimizations to happen on the list only.

5.4 Static Single-Assignment Form (SSA Form)

Static Single-Assignment Form. An IR that facilitates certain code optimizations.

- In SSA, each name receives **a single assignment only**.
  - The same variable will be renamed with different subscripts on multiple occurrences.
- **The φ function**. In multiple control-flow paths, the same variable may be combined using the φ-function.

> **if** (flag) $x_1 = -1$; **else** $x_2 = 1$;
> $x_3 = \varphi(x_1, x_2)$;

5.5 Types and Type Checking

Data type or simply *type* tells a compiler or interpreter how the programmers intend to use the data.

The usefulness of type information:

- Verify type expressions and code validity.
- **Storage allocation**.
- **(Array/Access) Offset computation**
- **Type conversions**
- **Operator overloading**: e.g., choose `fadd` for floats and `iadd` for integers.

Type Checking. Uses logical rules to make sure that the types of the operands match the type expectation by an operator.

5.5.1 Type Expressions

A **type expression** is one of the following:

1. A basic type
   - `boolean, char, integer, float, void, ...`
2. A type name (e.g., **name of a class**)
3. A type constructor applied to type expressions

- `array(Size, ElementType)`
- `record([Name1, Name2, ...], [Type1, Type2, ...])`
- `->` for function types (return type)

5.5.2 Type Equivalence

5.5.2.1 Named Equivalence

Treat **name types** as **basic types**; Names in type expression are not replaced by the exact type expressions they define.

- Two type expressions are name equivalent IFF **they are identical**.

```
typedef struct {
    int data[100];
    int count;
} Stack;
```

```
typedef struct {
    int data[100];
    int count;
} Set;
```

```
Code under analysis:
Stack x, y;
Set r, s;
x = y;   ✓
r = s;   ✓
x = r;   ✗
```

5.5.2.2 Structural Equivalence

For named types, **replace** the names by the **type expressions** and **recursively check the substituted trees**.

5.5.3 Declarations

$$
\begin{array}{lcl}
D & \rightarrow & T\ \textbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \textbf{int} \mid \textbf{float} \\
C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C
\end{array}
$$

- Nonterminal D generates a sequence of declarations.
- T generates basic, array or record types,
  - A record type is a sequence of declarations for the fields of the record, surrounded by the curly braces.
- B generates one of the basic types: `int` and `float`.
- C generates sequences of one or more integers, each surrounded by brackets.

5.5.4 Storage Layout for Local Names

- The **width** of a type: number of memory units needed for an object of the type, including **pointers**.
- For **local names of a function**, we always assign contiguous bytes
  - Compute **relative address** at **compile time**
  - Type info and rel-addr stored in **symbol table**

5.5.5 An SDT for Computing Types and Their Widths

$$
\begin{array}{lll}
T \rightarrow B & & \{\ t = B.type;\ w = B.width;\ \} \\
\quad C & & \{\ T.type = C.type;\ T.width = C.width;\ \} \\
B \rightarrow \textbf{int} & & \{\ B.type = integer;\ B.width = 4;\ \} \\
B \rightarrow \textbf{float} & & \{\ B.type = float;\ B.width = 8;\ \} \\
C \rightarrow \epsilon & & \{\ C.type = t;\ C.width = w;\ \} \\
C \rightarrow [\ \textbf{num}\ ]\ C_1 & & \{\ C.type = array(\textbf{num}.value,\ C_1.type); \\
 & & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

5.6 Type Checking

- Assign a *type expression* to each component of the source program
- Determine whether the type expressiosn conform to a collection of logical rules

5.6.1 Sound Type System

A **sound** type system allows us to determine **statically** that types errors cannot occur at runtime.

- A language is **strongly typed** if the compiler guarantees that the program it accepts will run without type errors.
  - **Strongly typed**: Java
  - **Weakly typed**: C/C++ (with *implicit conversion*)

5.6.2 Type Synthesis

Build up the type of an expression from the types of subexpressions.

- *Example.* Function type `f(int, int) -> int`

5.6.3 Type Inference

Determine the type of a language construct from the way it is used

- *Example.* Deduce argument expression type from that of the parameter. `f(float) -> int` and `f(x)` implies `x` is of type `float`.

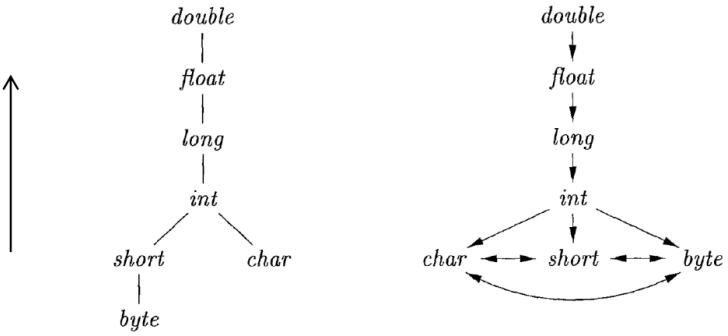
5.6.4 Type Conversion

- $E \rightarrow E_1 + E_2$

$$
\begin{array}{l}
\{\quad \textbf{if}(E_1.type = integer\ \textbf{and}\ E_2.type = integer)\ E.type = integer; \\
\quad\ \ \textbf{else if}(E_1.type = float\ \textbf{and}\ E_2.type = integer)\ E.type = float; \\
\quad\ \ \ldots \\
\}
\end{array}
$$

Figure 44: SDT for a simple case.

5.6.4.1 Widening and Narrowing



(a) Widening conversions (b) Narrowing conversions

- **Widening** conversions preserve information, and can be automatically done by the compiler. (*Implicit*, *Coercions*)
- **Narrowing** conversions may lose information and require programmers to write code to cause the conversion. (*Explicit*, *Casts*)
- *Example functions.* `max` return the upper bound of two types. Similar for `min`.

5.7 Translation Process

5.7.1 Translation of Declarations and Offset Computation

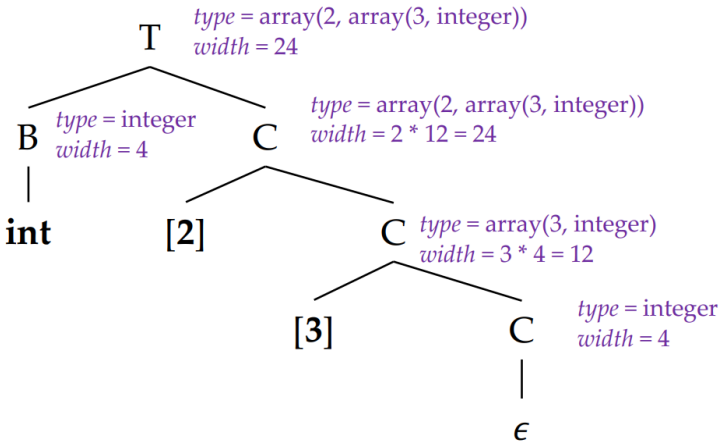


Figure 46: Computing the `width` information of types

$$
\begin{array}{lll}
P \rightarrow & & \{\ offset = 0;\ \} \\
\quad D & & \\
D \rightarrow T\ \textbf{id}\ ; & & \{\ top.put(\textbf{id}.lexeme,\ T.type,\ offset); \\
 & & \quad offset = offset + T.width;\ \} \\
\quad D_1 & & \\
D \rightarrow \epsilon & &
\end{array}
$$

Figure 47: Computing the `offset` information of types

5.7.2 Expression Translation

PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \mid\mid gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \mid\mid gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

- `S.code` and `E.code` denote three-address code.
- `E.addr` denotes the address that will hold the value of E.
- `top` denotes the current symbol table; `get` returns the address of `id` (a variable)
- `gen` generates three-address instructions

5.7.3 Incremental Translation Scheme

In the previous figure, let `gen` also **append the generated instruction to a global list of instructions generated so far**.

- The order is guaranteed when doing preorder traversal, which guarantees the order of accessing children (thus nodes in the frontier).

5.7.4 Computing Array Offset

Assume array elements are stored contiguously.

Suppose the array offset vector is $\boldsymbol{w} = (w_1\ \cdots\ w_n)^T$, the index vector is $\boldsymbol{i} = (i_1\ \cdots\ i_n)^T$ where i_n is the last index in access pattern `arr[i_1]...[i_n]`. Then the total offset is given by base $+\ \boldsymbol{w}^T\boldsymbol{i} \times$ base type width.

5.7.4.1 Computing Array Reference

- May use the type expression `array(Size, ElementType)` to compute the offsets of a sub-array in **an array of arrays**.

$S \rightarrow \textbf{id} = E\ ;$ $\quad \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \}$

$\mid\ L = E\ ;$ $\quad \{\ gen(L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \}$

$E \rightarrow E_1 + E_2$ $\quad \{\ E.addr = \textbf{new}\ Temp();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \}$

$\mid\ \textbf{id}$ $\quad \{\ E.addr = top.get(\textbf{id}.lexeme);\ \}$

$\mid\ L$ $\quad \{\ E.addr = \textbf{new}\ Temp();\ gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \}$

$L \rightarrow \textbf{id}\ [\ E\ ]$ $\quad \{\ L.array = top.get(\textbf{id}.lexeme);\ L.type = L.array.type.elem;\ L.addr = \textbf{new}\ Temp();\ gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \}$

$\mid\ L_1\ [\ E\ ]$ $\quad \{\ L.array = L_1.array;\ L.type = L_1.type.elem;\ t = \textbf{new}\ Temp();\ L.addr = \textbf{new}\ Temp();\ gen(t\ '='\ E.addr\ '*'\ L.type.width);\ gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}$

Figure 48: Semantic actions for array references

- `L.array`: a pointer to the symbol-table entry for the array name
- `L.array.base`: the base address of the array
- `L.addr`: a temporary for computing the offset for the array reference

5.8 Control Flow

5.8.1 Short-Circuiting of Conditional Expressions

The boolean operators `&&`, `||`, `!` are translated into jumps.

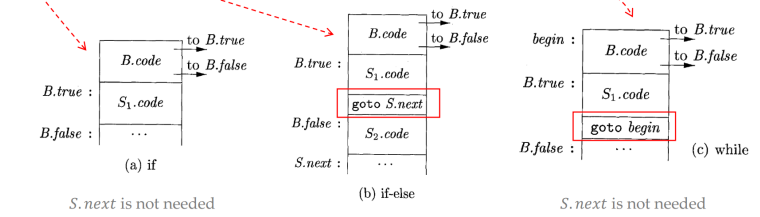
- Such that the *short-circuiting* property is satisfied.

5.8.2 Flow-of-Control Statements

- Grammar:
  - $S \rightarrow \textbf{if}\ (\ B\ )\ S_1$
  - $S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$
  - $S \rightarrow \textbf{while}\ (\ B\ )\ S_1$

Inherited attributes:

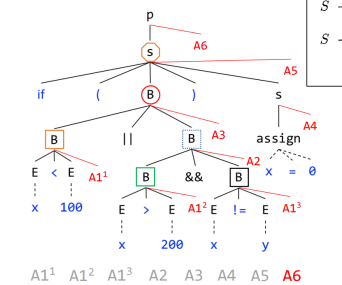
- $B.true$: the label to which control flows if B is true
- $B.false$: the label to which control flows if B is false
- $S.next$: the label for the instruction immediately after the code for S



$S.next$ is not needed $\qquad$ $S.next$ is not needed

Example

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$ A6
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$ A4
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ A5 $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ A3 $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1\ \&\&\ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ A2 $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow E_1\ \textbf{rel}\ E_2$	$B.code = E_1.code \mid\mid E_2.code$ A1 $\mid\mid gen('\textbf{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\textbf{goto}'\ B.true)$ $\mid\mid gen('\textbf{goto}'\ B.false)$



Generated code:

```
if x < 100 goto B.true = L2
goto B.false = L3
L3: if x > 200 goto B.true = L4
    goto B.false = B.false = B.false = S.next = L1
L4: if x != y goto B.true = B.true = B.true = L2
    goto B.false = B.false = B.false = S.next = L1
L2: x = 0
L1: ...
```


Figure 50: Short circuiting
for expression `if (x < 100 || x > 200 && x != y) x = 0;`

5.9 Backpatching

Key problem. Match a jump instruction with the jump target.

- When a jump is generated, its target is temporarily left unspecified.
- Incomplete jumps are grouped into lists. All jumps on a list have the same target.
- **Fill in the labels** for incomplete jumps **when the target becomes known**.

Techniques.

- `truelist`/`falselist`, whose target is the jump target when B is true/false
- `makelist(i)`, create a new list containing only i, the index of a jump instruction, and return the pointer to the list.
- `merge(p1, p2)`, concatenate the lists pointed by p_1 and p_2, and return a pointer to the concatenated list.
- `backpatch(p, i)`, insert i as the target for each of the jump instructions on the list pointer by p.

5.9.1 SDT for Boolean Expresson Generation during Bottom-Up Parsing

$B \rightarrow B_1 \parallel MB_2 \mid B_1 \&\& MB_2 \mid !B_1 \mid (B_1) \mid E_1\ \textbf{rel}\ E_2 \mid \textbf{true} \mid (\textbf{false})$

1) $B \to B_1$ || M B_2 { $backpatch(B_1.falselist, M.instr)$;
$B.truelist = merge(B_1.truelist, B_2.truelist)$;
$B.falselist = B_2.falselist$; }

2) $B \to B_1$ && M B_2 { $backpatch(B_1.truelist, M.instr)$;
$B.truelist = B_2.truelist$;
$B.falselist = merge(B_1.falselist, B_2.falselist)$; }

3) $B \to$! B_1 { $B.truelist = B_1.falselist$;
$B.falselist = B_1.truelist$; }

4) $B \to$ (B_1) { $B.truelist = B_1.truelist$;
$B.falselist = B_1.falselist$; }

5) $B \to E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr)$;
$B.falselist = makelist(nextinstr + 1)$;
$gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_')$;
$gen('\texttt{goto}\ \_')$; }

6) $B \to$ **true** { $B.truelist = makelist(nextinstr)$;
$gen('\texttt{goto}\ \_')$; }

7) $B \to$ **false** { $B.falselist = makelist(nextinstr)$;
$gen('\texttt{goto}\ \_')$; }

8) $M \to \epsilon$ { $M.instr = nextinstr$; }

- M is used to emit short-circuting labels. It pick up, at appropriate time, the index of the next instruction to be generated.
  - This marker nonterminal M produces, as a synthesized attribute $M.instr$, the index of the next instruction, just before B_2 code starts being generated.

$$M.instr = nextinstr;$$

5.9.2 Backpatching vs Non-backpatching

1. Non-backpatching SDD with inherited attributes
2. Backpatching schemes: maintaining lists

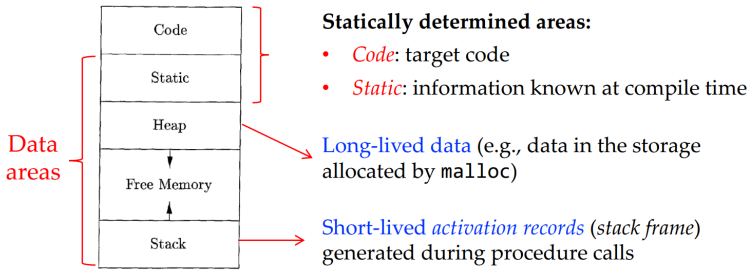
The assignments to true/false attributes in (1) correspond to the manipulations of `truelist`/`falselist` in (2).

6 Runtime Environments

Compilers manage a *runtime environment*, and deals with:
- Layout and allocation of storage locations for data in the source program
- Mechanisms to access variables
- **Linkages** between procedures, the mechanisms for passing parameters

6.1 Storage Organization

Runtime memory can be typically divided into **code** and **data areas**.



- **Static Allocation**. The allocation decision can be made by the compiler by looking only at the program text.
  - Global constants, global variables
- **Dynamic**. A decision can be made only while the program is running.
  - *Stack Storage*. The space for names local to a procedure is allocated on a stack. The lifetime of the data is the same as that of the called procedure.
  - *Heap Storage*. Hold data that may *outlive* the call to the procedure that created it.
    - Manual Memory Deallocation
    - Automatic Memory Deallocation, aka Garbage Collection

6.2 Stack Space Allocation

6.2.1 Activation Tree

The activations of procedures during the running of an entire program can be represented by an **activation tree**.

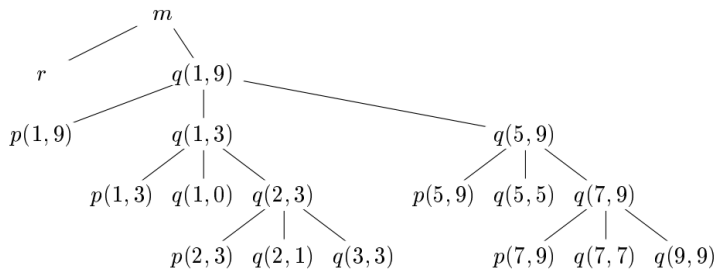


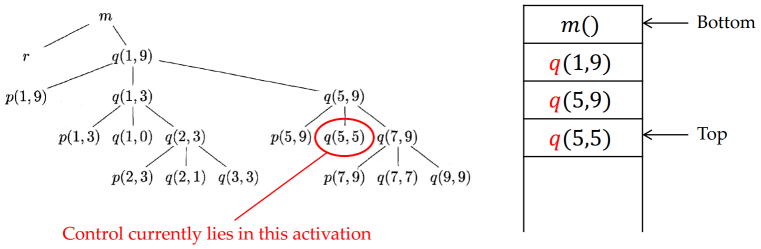
Figure 53: m represents the `main` function, r represents the `readarray()` function, and $q(\text{start}, \text{end})$ represents `quicksort(start, end)`.

6.2.2 Activation Records

6.2.2.1 Control Stack (Call Stack)

Procedure calls and returns are usually managed by a runtime stack called the *control stack* (or *call stack*).

Each live activation has an **activation record** (or **stack frame**) on the control stack.



6.2.2.2 Layout of an Activation Record

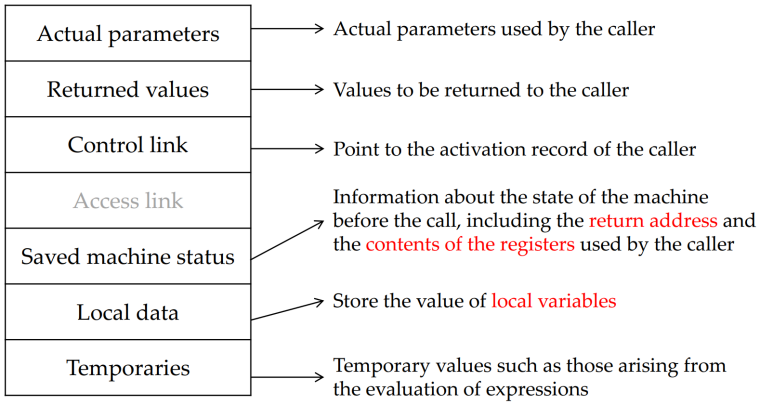


Figure 55: In JVM, the default size of the call stack for each thread is 1MB.

- **Values passed between caller and callee** are put at the beginning of the callee's activation record (next to the caller's record)
- **Fixed-length items** (control link, access link, save machine status) are in the middle
- **Items whose size may not be known** early enough are placed at the end
- The **top-of-stack** (*top_sp*) pointer points to the end of the fixed-length fields.

6.2.3 Calling Sequence

A calling sequence consists of code that:
1. allocates an activation record on the stack, and
2. enters information into the fields of that activation record.

6.2.3.1 Return Sequence

A return sequence restores the state of the machine so that the caller can continue its execution after the call.

6.2.3.2 Responsibilities in the Calling Sequence

There is **no exact division** between the caller and the callee in a calling sequence.
- **Dependencies**: Source language, target machine, OS, ...
- **General Rule**: Put as much code as possible into the **callee**.

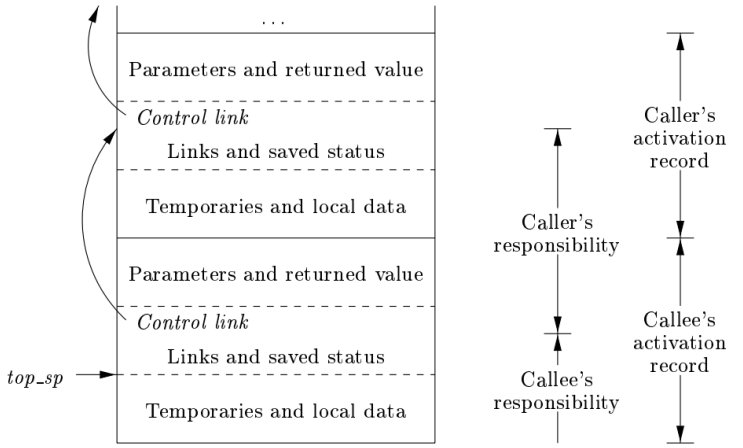
6.2.4 Composition of A Calling Sequence

1. The caller evaluates the **actual parameters**
2. The caller stores the return address and the old value of *top_sp* into the callee's activation record.
3. The caller increments *top_sp* accordingly.
4. The **callee** saves the **register values** and other **status info**

5. The **callee** initializes **its local data** and begins execution.

6.2.5 Composition of A Return Sequence

1. The Callee places the **return value** next to the actual parameters fields in its activation record.
2. Using information in the machine-status field, the callee restores *top_sp* and other registers.
3. Go to the **return address** set by the caller.



Why store top_sp? Because there is no other way for the caller to know where these data are stored.

6.3 Heap Management

The heap is used for data that **lives indefinitely**, or until the porgram explicitly deletes it.

- `new`, `malloc`, `free`, ...

6.3.1 The Memory Manager

Memory manager *allocates and deallocates space within the **heap***.

- Serves as an *interface* between application programs and the OS

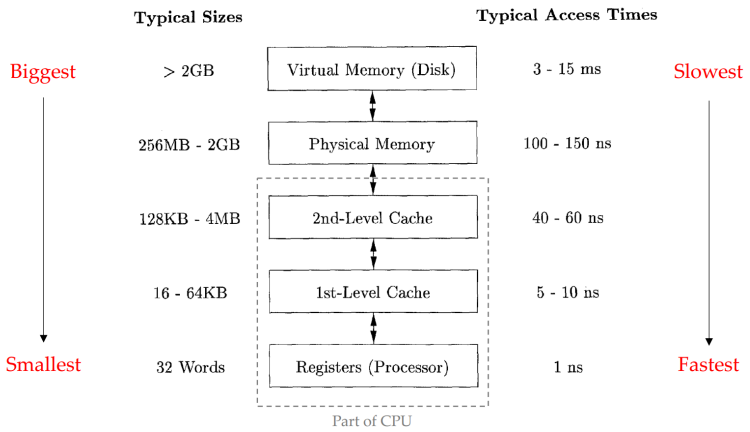
6.3.1.1 Basic Functions

- **Allocation**. Allocate a *contiguous heap memory*. If possible, use free space in the heap. Otherwise, get *virtual memory* to increase heap storage space from the OS.
- **Deallocation**. Return deallocated space to the pool of free space for reuse.
  - Typically, memory managers do not return memory to the OS, even if the program's heap usage drops.

6.3.1.2 Desirable Properties

- **Space Efficiency**. Minimize fragementation.
- **Program Efficiency**. Organize memory hierarchy to allow programs run faster.
- **Low Overhead** on memory allocation/deallocation.

6.3.2 Memory Hierarchy



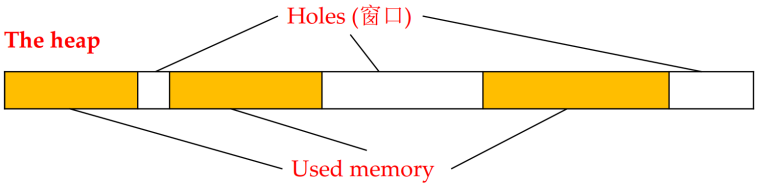
6.3.3 Program Locality

- **Temporal Locality**. The memory locations accessed are likely to beaccessed again *within a short period of time.*
- **Spatial Locality**. Memory locations close to the locations accessed are likely to be accessed *within a short period of time.*
- **Pareto's Principle**. Programs spend **90%** of their time executing **10%** of the code. (or 80%/20%).

Place frequently used instructions/data on faster storage.

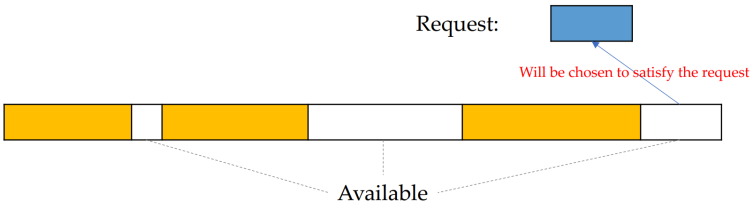
6.3.4 Fragmentation

As the program allocates/deallocates memory, the originally contiguous heap is broken up into free and used chunks.



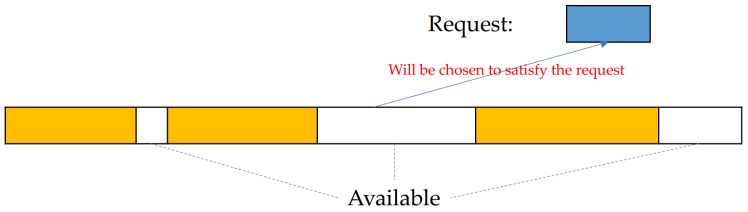
- The hols can only **get smaller** if no combination of contiguous holes or defragementation is to be done.

6.3.4.1 Best-Fit Algorithm



- Allocate the requested memory in the smallest available hole that is large enough
- **Improve space utilization** by sparing the large holes to satisfy subsequent larger requests.

6.3.4.2 First-Fit Algorithm

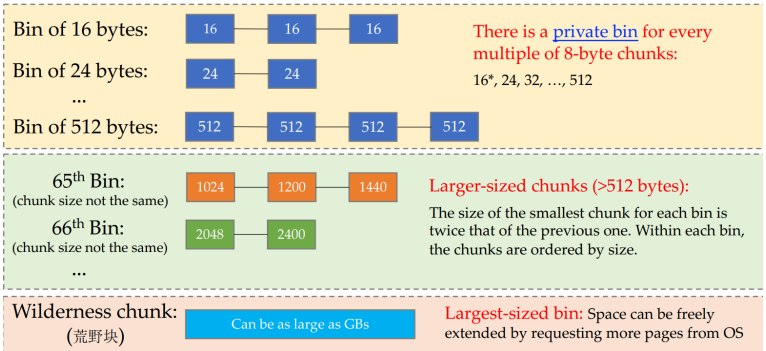


- Place in the first (lowest-address) hole in which it fits
- **Takes less time, improves spatial locality**. But inferior to best-fit in overall performance.

6.3.4.3 The Binning Strategy for Best-Fit

- Organize free chunks into **separated *bins***, according to chunk sizes.
- Have more bins for smaller-sized chunks (objects are often small)

6.3.4.4 The Doug Lea's Strategy



The **Lea memory manager** of the GNU C compiler `gcc` aligns all chunks to 8-byte boundaries.

- For **small size** requests (e.g., 48 bytes), if there is a bin for chunks of that size, we may take any chunk from the bin. The chunks are of the same size.
- For **sizes that do nto have a private bin** (e.g., 1224 bytes), we find the bin that is allowed to include chunks of the size.
  - Within the bin we may use any algorithm such as first-fit/best-fit.
- When all of the above fail, we continue to check.
- We may eventually reach wilderness chunk, in which the MM allocates space by **requesting more pages from the OS**.

7 Code Generation

7.1 Code Generator

- **Input**: IR + Symbol Table
- **Output**: Target Program

There is often an *optimization phase* before code generation.

Primary Tasks:

- Instruction Selection
- Register Allocation and Assignment
  - **Allocation**. What values should reside in registers?
  - **Assignment**. Which register should be used?
- Instruction Reordering

7.2 Design Issues

- **Design Goals**.
  - Correctness
  - Ease of implementation, testing and maintenance.
- **Many choices for the input IR**.
  - *Three-Address Representations*. Quadruples, triples, indirect triples.
  - *VM Representations*. Bytecodes and stack-machine code.
  - *Graphical Representations*. Synatx trees and DAG's.
- **Many possible target prorgams**.
  - RISC, CISC ({reduced, complex} instruction set computer), ...
  - Absolute machine-language programs;
  - Relocatable machine-language programs (**relative addresses only**, requires linking)
  - Assembly-language programs
  - ...

7.3 The Target Language

7.3.1 A Simple Target Machine Model

Type	Form	Effect
Load	LD $dst, addr$	load the value in location $addr$ into location dst, where dst is often a register
Store	ST x, r	store the value in register r into the location x
Computation	$OP\ dst, src_1, src_2$	apply the operation OP to the values in locations src_1 and src_2, and place the result in location dst
Unconditional jumps	BR L	jump to the machine instruction with label L
Conditional jumps	B$cond$ r, L	jump to label L if the value in register r pass the test B$cond$, e.g., less than zero

7.3.2 Addressing Modes

- **Direct Addressing**.
  - **Register.** By variable name, e.g., `x`.
  - **Immediate constant addressing**. `LD R1, #100`
- **Indirect Addressing**. By offsets to a location value, e.g., taking the l-value of a and adding to it the value in register r, `a[r]`.
  - **1-level indirect.**
    - `LD R1, 100(R2)`. Equivalent to $R_1 = \text{contents}(100 + \text{contents}(R_2))$
  - **2-level indirect.**
    - `LD R1, *100(R2)`. Equivalent to
      $$R_1 = \text{contents}(\text{contents}(100 + \text{contents}(R_2)))$$
    - `LD R1, * R2`. Equivalent to $R_1 = \text{contents}(\text{contents}(\text{contents}(R_2)))$

7.4 Addresses in the Target Code

7.4.1 Handling Procedure Calls and Returns

7.4.1.1 Static Allocation

The **size** and **layout** of activation records are determined by the code generator via the information in the symbol table.

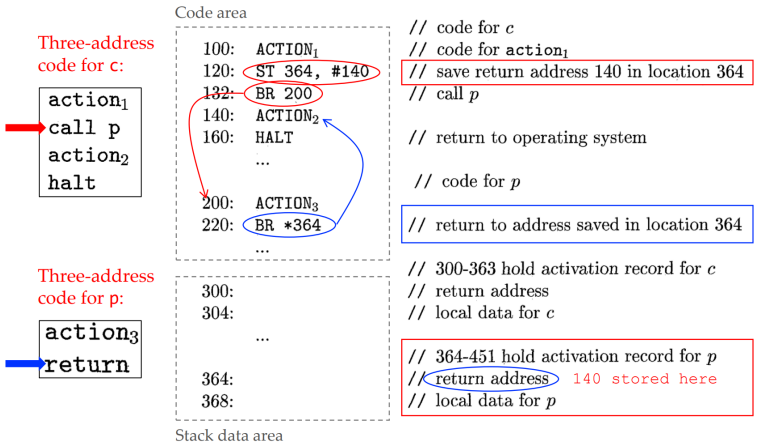
- Target program code for the three-address code: call *callee*

```
ST   callee.staticArea, #here + 20
BR   callee.codeArea
```

Store the return address (the address of the instruction after BR) at the beginning of the callee's activation record in the *Stack* area of the run-time memory[1,2]

codeArea gives the address of the first instruction of the *callee* in the *Code* area of the run-time memory

- `staticArea` gives the address of the beginning of an activation record.
- Three constants + 2 instructions results in 5 words, thus the offset 20.
- *Here we choose to store the return address directly at the beginning of the record.*
- Code for `return` now becomes `BR *callee.staticArea`, transferring control to the next instruction after the function invocation.



7.4.1.2 Stack Allocation

Static allocation uses absolute addresses, which is not suitable for real cases.

Using **relative addresses** solves the problem.

- Maintain in a register `SP` a pointer to the beginning of the activation record on top of the stack.
  - A procedure calling sequence

Each takes 4 bytes

```
ADD  SP, SP, #caller.recordSize   // increment stack pointer
ST   *SP, #here + 16              // save return address*
BR   callee.codeArea              // jump to the callee
```

Additional work comparing to static allocation

  - The return sequence

```
BR   *0(SP)                       // return to caller (done in callee)
SUB  SP, SP, #caller.recordSize   // decrement stack pointer (done in caller)
```

Additional work comparing to static allocation

7.4.2 Handling Names in Intermediate Code

A **name** in a three-address statement corresponds to a symbol-tabel entry.

Assume statement `x = 0`. Assume an offset of `12`.

- **Statically allocated area**. `LD 112, #0`. (Suppose static area starts at address `100`).
- **In an activation record**. `LD 12(SP), #0`.

7.5 Basic Block and Flow Graph

7.5.1 Basic Blocks

The intermediate code is partitioned into *basic blocks*.

- The control flow can only enter the basic block through **its first instruction**.
- Control will not leave the block, except possibly at the last instruction in the block. (**no halting/branching in the middle**).

7.5.1.1 Partitioning Three-Address Instructions into Basic Blocks.

Input. A sequence of three-address instructions
Output. A list of basic blocks (each inst. is assigned to one block).
Method.

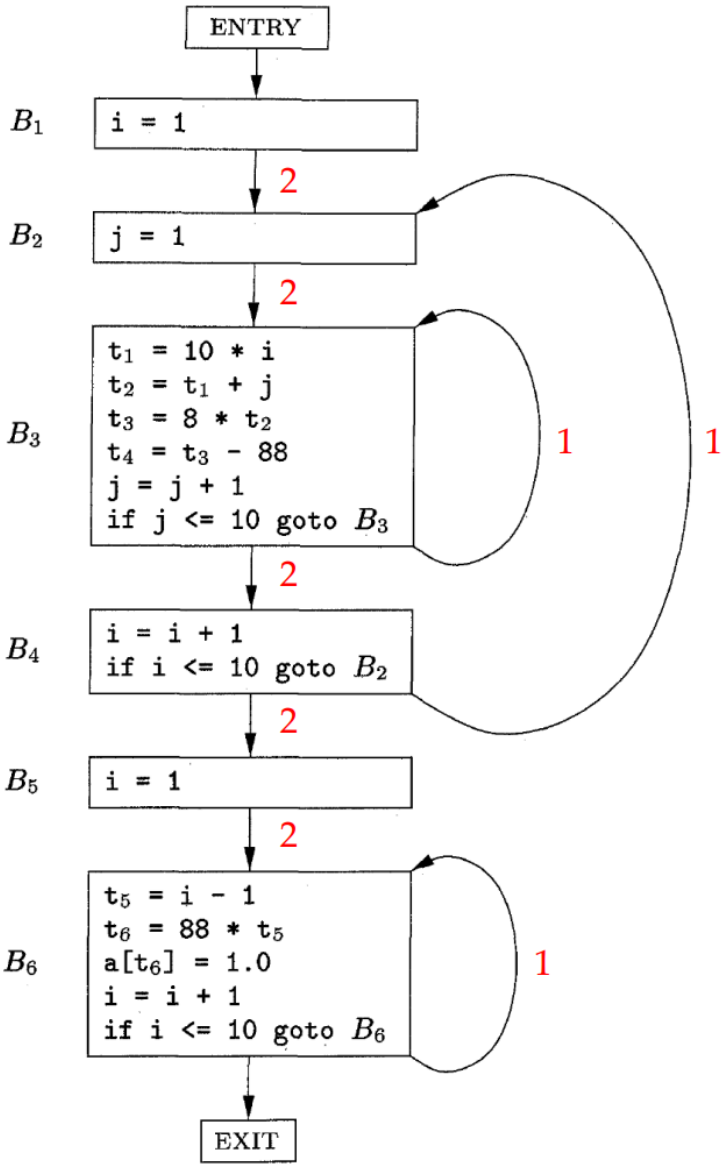
- Find the *leader instructions*. (The 1st instruction of a basic block)
  - The **first instruction** in the entire intermediate code is a leader
  - Any instruction that is the **target of a (un)conditional jump** is a leader
  - Any instruction that **immediately follows a (un)unconditional jump** is a leader.
- Then, for each leader, its basic block consists of **itself and all instructions up to but not including the next leader** or the end of the intermediate program.

7.5.2 Flow graphs

Basic blocks are *nodes*, and the *edges* indicate which block can follow which other blocks.

Flow graphs form the **basis** of code optimization.

- They describe how control flows among basic blocks.
- We can know how values are defined and used.



- **Entry** and **exit**:
  - Do not correspond to executable instructions
  - There is an edge from the entry to the first executable node
  - There is an edge to the exit from **any basic block** that could be executed last.

7.5.3 Loops

A loop L is a set of nodes in the flow graph.

- L contains a node e called the *loop entry*.
- No node in L except e has a predecessor outside L. That is, every path from the entry of the entire flow graph to any node in L goes through e. (e dominates the other nodes in L).
- Every node in L has a **nonempty** path, **completely within** L, to e.

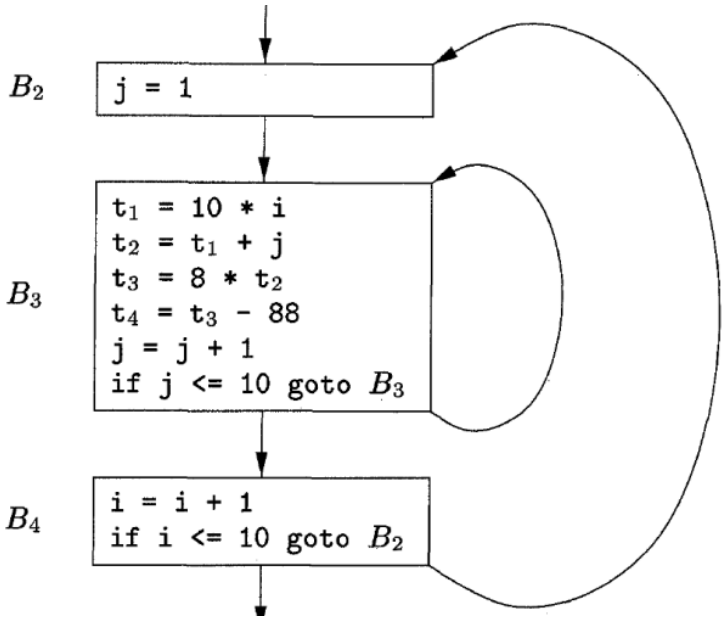


Figure 64: B_2 is the loop entry.

7.6 Optimization of Basic Blocks

7.6.1 The DAG Representation of Basic Blocks

Depicts the relationships among the values of all variables in a basic block when it executes. (**data dependence**)

7.6.1.1 Creating a DAG

1. Create a node for each of the initial values of the variables in the basic block.
2. Create a node N for each statement within a block. The children of N are those nodes corresponding to statemeents that are the **last definitions**, prior to s, of the operands used by s.
3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the **last definition** within the block.
4. Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exist* from the block; That is, is their values may be used later, in another block of the flow graph. Calculation of these **live variables** is a matter for global flow analysis. (Textbook 9.2.5)

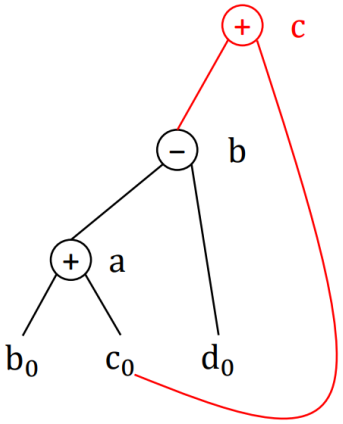
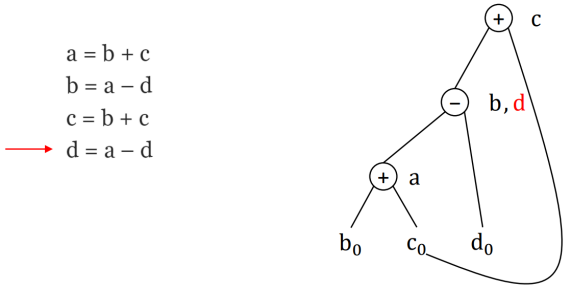


Figure 65: DAG for `a = b + c; b = a - d; c = b + c;`

7.6.2 Finding Local Common Subexpression

When creating a node M, check if there exists a node N, which has the same operator and children ndoes (**order matters**) with M. If such a node N exists, we do not create the node M, but simply **use** N **to represent** M.

- **Must point to the same nodes in the DAG** (same last definition).



7.6.3 Dead Code Elimination

We can delete from a DAG any **root** (node without ancestors) that has **no live variables attached**. Repeatedly applying such transformation will remove all nodes corresponding to dead code.

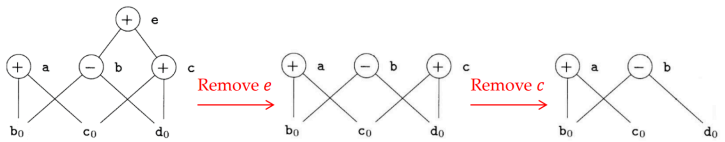


Figure 67: Suppose only a and b are live.

7.6.4 Algebraic Identities

- **Eliminate computations**.
  - $x + 0 = 0 + x = x \qquad x - 0 = x$
  - $x \times 1 = 1 \times x = x \qquad x/1 = x$
- **Reduction in strength**. Replacing a more *expensive* operator by a *cheaper* one.
  - $2 \times x = x + x \qquad x/2 = x \times 0.5$
- **Constant folding**.
  - $2 \times 3.14 = 6.28$

7.7 Code Generator

Primary Goal. Avoid redundant loads/stores.

- **Making the best use of registers**.

Using registers:

- Hold operands to perform operations
- Hold temporaries
- Hold global values
- Help with runtime storage management (e.g., stack pointers).

7.7.1 Basic Process

1. **Load**..
2. **Computation**.
3. **Store**.

7.7.2 Descriptors

- **Register Descriptor**. For each available register, keeping track of the variable names whose current value is in that register.
- **Address Descriptor**. For each program variable, keeping track of the locations where the current value of that variable can be found.
  - A **location** may be a register, a memory address, a stack location, ...

7.7.3 The Algorithm

7.7.3.1 Utilities

- **`getReg(I)`. Select registers for aech memory location** associated with the three-address instruction I, according to the descriptors and data-flow info (live variables on block exit)

7.7.3.2 Generation

- **For an arithmetic TAI**:
  1. Select a register use `getReg(I)`. If the operands are not present, generate a **load instruction**.
  2. Generate `res = t1 op t2`
- **For a copy instruction** $x = y$:
  1. Assume `getReg(I)` will always **select** the same registers for x and y.
  2. If y is not in that register R_y, then generate an instruction `LD Ry, y`
- **Ending a Basic Block**
  - **For temporary variables**, just don't care
  - **If a varaible is live on block exit**, generate `ST` if its latest value (from the address descriptor) is not currently in its desired memory location.

7.7.3.3 Updating Descriptors

1. For the instruction `LD R, x`
   1. Change the **register descriptor** for R so it holds only x.
   2. Change the **address descriptor** for x by adding R as an additional location
   3. Remove R from the **address descriptor** of any variable other than x.
2. For the instruction `ST x, R`
   1. Change the **address descriptor** for x to include its own memory location.
3. For an operation such as `ADD Rx, Ry, Rz`
   1. Change the **register descriptor** for R_x so it holds only x
   2. Change the **address descriptor** for x so that its only location is R_x
   3. Remove R_x from the **address descriptor** of any variable other than x.
4. When processing a copy statement `x = y`
   1. If `LD Ry, y` is generated, manage descriptors using rule 1.
   2. Add x to the **register descriptor** for R_y
   3. Change the **address descriptor** for x so that its only location is R_y.

7.7.4 Design of the Function `getReg`

- **Goal**: Avoid too much data exchange with memory
- **Task**: **Pick registers** for **operands** and **result** of each three-addr inst.

For the generic example, $x = y + z$, pick register R_y for operand y.

- **If y is not in register.** Let R be a candidate and suppose v is a variable in R's register descriptor.
  - It is **safe** to reuse if:
    - v's address descriptor says we can find v somewhere besides R
    - v is x and x is not the other operand z
    - v is not used later.
  - Otherwise, generate `ST v, R` to place a copy of v in its own memory location (spill)
- If R holds multiple varaibles, repeat the process for each such variable v
  - Pick the one that generates the lowest number of `ST`.

7.7.5 Principles of Register Allocation

- Assign registers to frequently used variables, and keep these registers consistent across block boundaries.
- **Estimate the benefit**. Requires quantitative analysis.

7.8 Optimziations

7.8.1 Peephold Optimization

- Maintaining a **sliding window**:
  - Redundant-instruction elimination
  - Flow-of-control optimizations
    - Unreachable cod

8 Data-Flow Analysis

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
Boundary	OUT[ENTRY] = ∅	IN[EXIT] = ∅	OUT[ENTRY] = ∅
Meet (∧)	∪	∪	∩
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,\, pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,\, succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,\, pred(B)} \text{OUT}[P]$
Intialize	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$
Application	Constant folding	Dead code elimination	Global common subexpression elimination