A Handbook on
Computer Science

7

Compiler Design

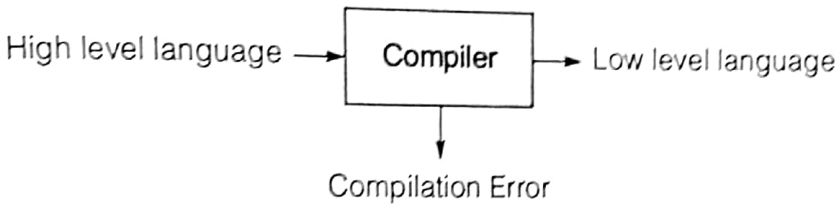
CONTENTS

Lexical Analysis

1

Introduction

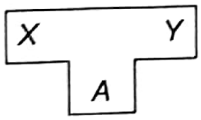
A program written in one high level language (source language) producing output in low level language (object or target language) is called compiler.



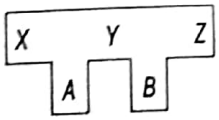
Cross Compiler

A compiler that runs on one machine '*A*' and produces a code for another machine '*B*'.

Example: (a) Compiler runs on machine *A*, which takes input language *X* and produces output language *Y*.



(b) Compiler runs on machine *A* and produces code *Y* for another machine *B*.



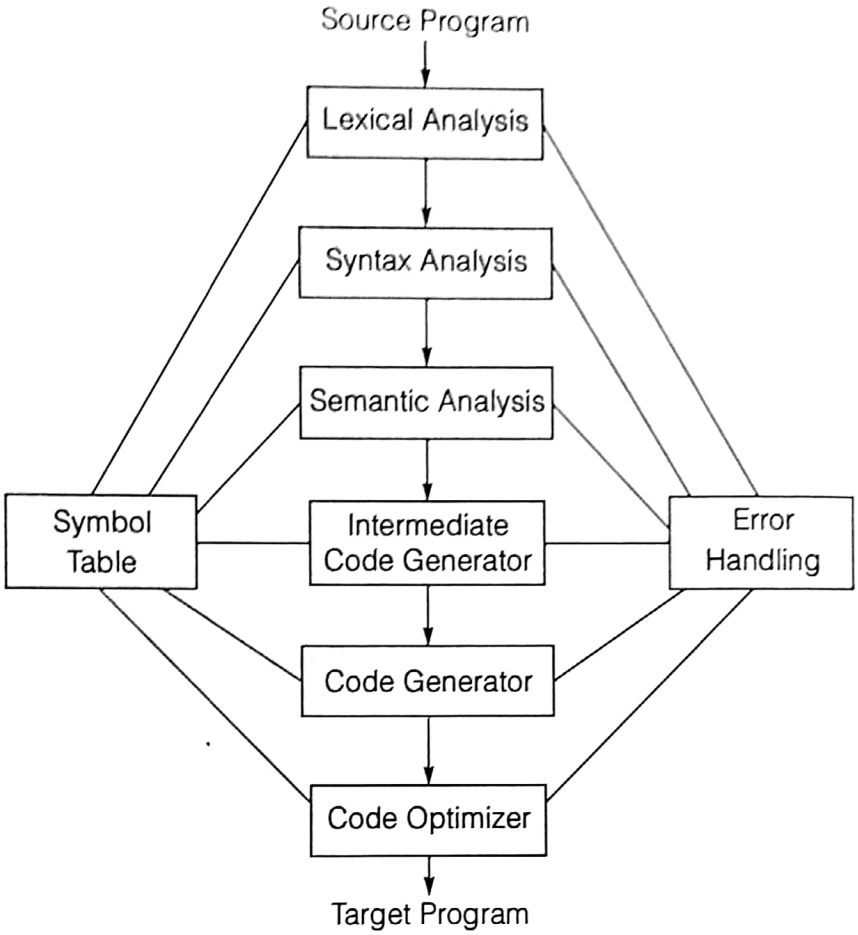
Compiler Parts

There are two major parts of a compiler:

1. **Analysis Phase:** An intermediate representation is created from the given source program
   (*i*) Lexical Analyzer
   (*ii*) Syntax Analyzer
   (*iii*) Semantic Analyzer
2. **Synthesis Phase:** Equivalent target program is created from intermediate representation
   (*i*) Intermediate Code Generator

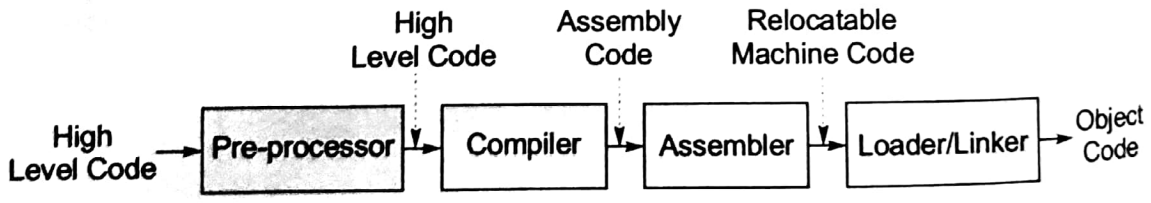
(*ii*) Code Generator

(*iii*) Code Optimizer



Compiler Phases

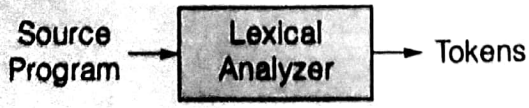
Language Processing System



Language Processing System Using Compiler

Lexical Analysis

- Lexical analyzer reads a source program character by character to produce tokens.



- A token can represent more than one lexeme, additional information should be held for that specific lexeme. This additional information is called as attribute of the token.

- Token type and its attribute uniquely identify a lexeme
- Regular expressions are used to specify patterns and finite automata is used to recognise tokens.
- Tokens identified by lexical analyzer: Keywords, constants, identifiers, operators and symbols.

Designing of Lexical Analyzer

- Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyser.
- Design a state diagram that describes the tokens and write a program that implements the state diagram.
- Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram.

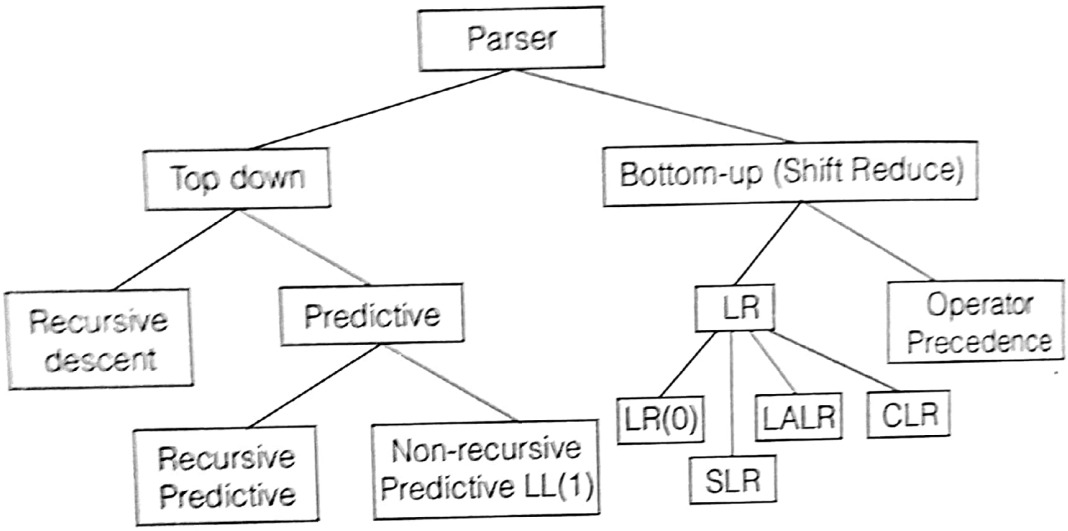
■■■

Parsing

2

Introduction

- Syntax analyzer is also known as parser.
- The syntax of a programming language is described by a context free grammar.
- A parser works on stream of tokens which are generated by lexical analyser.



Ambiguous Grammar

For a given input string if there exists 'more than one parse tree' then that grammar is called as "ambiguous grammar".

Left Recursion

The grammar : $A \rightarrow A\alpha | \beta$ is left recursive. Top down parsing techniques cannot handle left recursive grammars, so we convert left recursive grammars into right recursive.

Left Recursive Elimination

$$\text{Left recursive : } A \rightarrow A\alpha | \beta \Rightarrow \boxed{\begin{matrix} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' | \varepsilon \end{matrix}} \text{ (non left recursive)}$$

Problem with left recursive grammar is that if such a grammar is used by top down parser (uses Left Most Derivation), then there is a danger of going into "infinite loop" while string is parsing.

Left Factoring

- A predictive parser (a top down parser without backtracking) insists that the grammar must be left-factored.
- Left factoring avoids backtracking in parser but is not a solution for elimination of ambiguity.
- If a grammar has common prefixes in r.h.s. of non-terminal then such grammar need to be left factored by eliminating common prefixes as follows:

$$A \to \alpha\beta_1 | \alpha\beta_2 \Rightarrow \boxed{\begin{array}{l} A \to \alpha A' \\ A' \to \beta_1 | \beta_2 \end{array}} \quad \text{(left factored grammar)}$$

Parsers

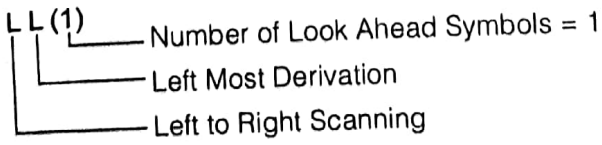
1. **Top-down parsers:** Starting at the root (the starting symbol) and proceeding to the leaves. This parsers are easy to write, actual code capable of being derived directly from the production rules, but cannot always applied as an approach.
2. **Bottom-up parsers:** Start at the leaves and move up towards the root. Bottom-up parsers, can handle a larger set of grammar.

Top Down Parsing (TDP)

1. **Recursive-descent parsing:**
   (*i*) Follows Brute Force mechanism (Suffers from Backtracking).
   (*ii*) Not efficient, general purpose, not widely used.
2. **Predictive parsing:**
   It can uniquely choose a production rule by just looking the current symbol.
   (*i*) Recursive predictive parsing: each non terminal corresponds to a procedure.
   (*ii*) Non-recursive predictive parsing: Table driven also called as LL(1) parser.

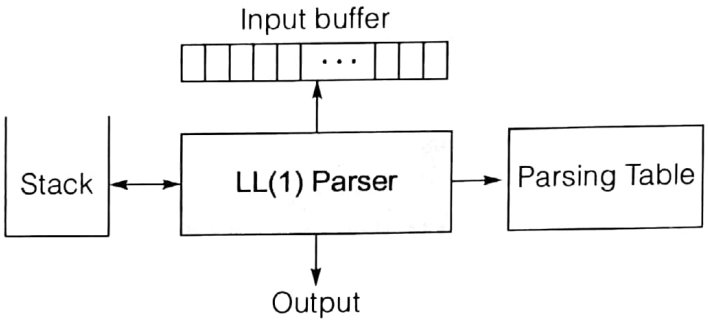
LL(1) Parser/Predictive Parser

- LL(1) grammar is unambiguous, left factored and non left recursive.
- Top down parser follows left most derivation.



- FIRST(A) is a set of the terminal symbols which occur as first symbols in strings derived from A.
- Follow(A) is the set of terminals which occur immediately after the nonterminal A in the strings derived from the starting symbol.

Configuration of LL(1) Parser



Configuration of LL(1) Parser

LL(1) Parsing Table Construction

- First( ) and Follow( ) sets are used to construct LL(1) parsing table.
- For each rule $A \rightarrow \alpha$ in grammar G:
  - (*i*) For each terminal 'a' contained in FIRST(A) add $A \rightarrow \alpha$ to [A, a] entry in parsing table if α derives '*a*' as the first symbol.
  - (*ii*) If FIRST(A) contain 'ε', for each terminal 'b' in FOLLOW(A), add $A \rightarrow \varepsilon$ to [A, b] entry in parsing table.

LL(1) Grammar

- To check whether given grammar is LL(1) or not:

  (*i*) If $A \rightarrow \alpha_1 | \alpha_2 \Rightarrow \{\text{FIRST}(\alpha_1) \cap \text{FIRST}(\alpha_2) = \phi\}$

  (*ii*) If $A \rightarrow \alpha | \varepsilon \Rightarrow \{\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \phi\}$

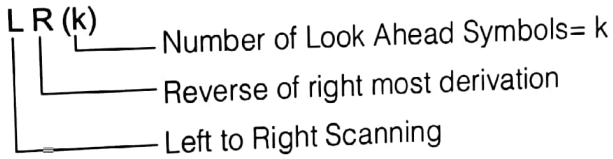
  } is LL(1)

- A grammar where its LL(1) parsing table has free from multiple entries is said to be LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar
- If a grammar is not left factored, it cannot be a LL(1)
- An ambiguous grammar cannot be a LL(1) grammar.

Bottom up Parsing

The process of constructing the parse tree starting from input string and reaching the start symbol of the grammar is known as Bottom Up Parsing.

- **Handle:** The part of the input string that matches with RHS of any production is called as handle.
- **Handle pruning:** The process of finding the handle and replacing the handle by LHS of corresponding production is called "handle priority".
- Bottom up parsing is also known as shift reduce parser
- Bottom up parsing can be constructed for both ambiguous and unambiguous grammar.
- For unambiguous grammar, LR parser can be constructed and Operator Precedence Parser (OPP) can be constructed for both ambiguous and unambiguous grammar.
- Bottom up parsing is more powerful than top down parser.
- Average time complexity is $O(n^3)$ and handle pruning is major overhead for bottom up parsing.
- Bottom up parser simulates the reverse of right most derivation.
- **LR(k) parser:**



Shift Reduce Parsers

Operator Precedence Parser (OPP)

- Constructed for 'operator precedence grammar', a grammar which do not contain epsilon productions and do not contain two adjacent non-terminals on *R.H.S.* of any production. Operator precedence parser grammar is provided with precedence rules.
- **Operator Precedence Parsing Algorithm:** Let '*a*' is top of stack symbol and '*b*' is symbol pointed by input pointer.
  - (*i*) If $a \doteq b \doteq \$$, successfully completion of parsing
  - (*ii*) If $a \lessdot b$ or $a \doteq b$ then push '*b*' onto stack and advance input points to next symbol.
  - (*iii*) If $a \gtrdot b$ then pop '*a*' and reduction takes place for '*a*'.
- Operator precedence grammar may be either ambiguous or unambiguous grammar

- Example for operator precedence table:

	id	+	*	$
id		$\dot{>}$	$\dot{>}$	$\dot{>}$
+	$\dot{<}$	$\dot{>}$	$\dot{<}$	$\dot{>}$
*	$\dot{<}$	$\dot{>}$	$\dot{>}$	$\dot{>}$
$	$\dot{<}$	$\dot{<}$	$\dot{<}$	

LR Parsers

- **LR(0) Parser:**
  - (*i*) LR(0) parser is constructed for LR(0) grammar in which the parse table is free from multiple entries (conflicts).
  - (*ii*) **Conflict in LR(0) parser:**
    - (a) **Shift reduce (SR) conflict:** When the same state in DFA contains both shift and reduced items.

      $A \to \alpha \cdot x\beta$ (shifting)

      $B \to \gamma \cdot$ (reduced)

    - (b) **Reduce Reduce (RR) conflict:**

      $A \to \alpha \cdot$ (reduced)
      $B \to \gamma \cdot$ (reduced)
      } Both are in same state of DFA

  - (*iii*) Closure( ) and goto( ) functions are used to create canonical collection of LR items.
  - (*iv*) LR(0) parsing table construction:
    - (a) If goto (I_i, a) = I_j, then set action [i, a] = S_j (shift entry)
    - (b) If goto (I_i, A) = I_j, then set action [i, A] = j (state entry)
    - (c) If I_i contains $A \to \alpha\cdot$ (reduced production) then action [i, all entries] = R_P, (reduced entry), where 'P' is production number ($P : A \to \alpha$)
    - (d) If $S' \to S\cdot$ is in I_i, then set action [i, $] to accept.
  - (*v*) If a state contains either *SR* or *RR* conflicts then that state is called 'Inadequate State'.
- **Simple LR(1) Parser:**
  - (*i*) **Conflicts in SLR(1) parser:**
    - (a) **Shift Reduce (SR) conflict:**

      $A \to \alpha \cdot x\beta$ (shift)

      $B \to \gamma \cdot$ (reduce)

If FOLLOW(B) $\cap \{x\} \neq \phi$

(b) **Reduce Reduce (RR) conflict:**

$A \to \alpha\cdot$

$B \to \gamma\cdot$ If FOLLOW(A) $\cap$ FOLLOW(B) $\neq \phi$

(*ii*) **SLR(1) parsing table construction:**

(a) SLR(1) parsing table construction is same as LR(0) except reduced entries.

(b) If I_i contains A $\to \alpha$ (reduced production), then find FOLLOW(A) and for every a in FOLLOW(A), set action $[i, a] = R_P$ {where P is production number}

(*iii*) SLR(1) is powerful then LR(0)

(*iv*) Size of SLR(1) parse table is same as size of LR(0) parse table.

(*v*) Every LR(0) grammar is SLR(1) but every SLR(1) need not be LR(0).

- **Canonical LR(1) or LR(1) Parser:**

(*i*) **CLR(1) parsing table construction:** Except reduced entries, remaining entries are same as SLR(1).

(*ii*) If I_i contains $A \to a\cdot, \$ \mid a \mid b$ then reduced entry (R_P) $A \to a$ in row 'i' under the terminals given by look ahead ($, a, b) in LR(1) items

[R_P entry entered into [i, $], [$i$, a] and [i, b]]

- **Look ahead LR(1) Parser:**

(*i*) It is constructed from CLR(1), if two states are having same productions but may contain different look aheads, those two states of CLR(1) are combined into a single state in LALR(1).

(*ii*) There is no chance of SR conflict but there may be chance for RR conflicts in LALR(1) after combining the states of CLR(1) parser.

(*iii*) Conflicts in CLR(1) and LALR(1)

(a) SR conflict: $A \to \alpha\cdot x\beta, a$; $B \to \gamma\cdot, x \mid y$

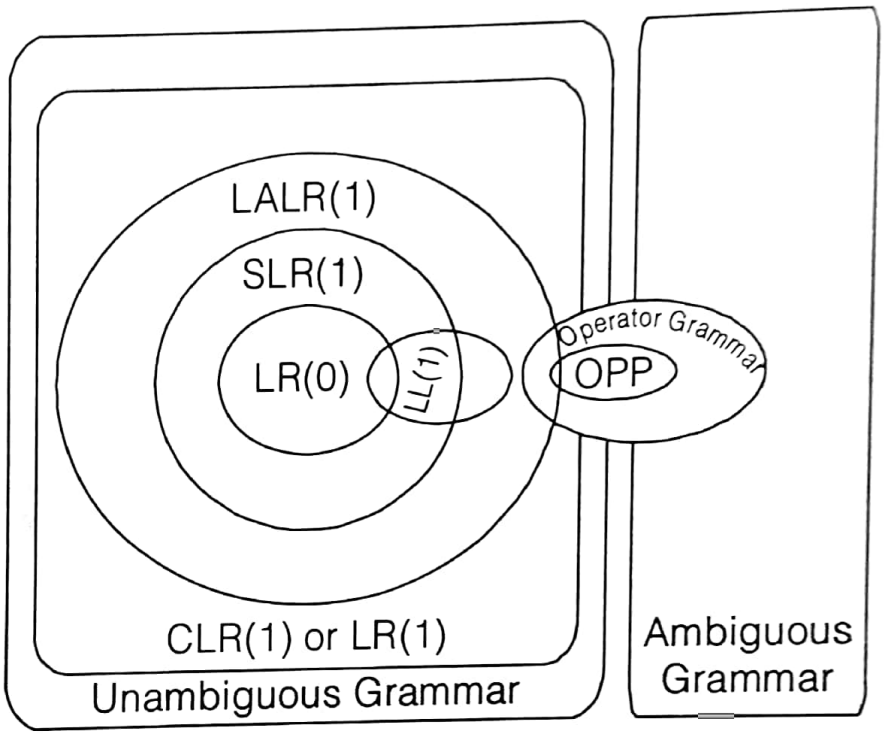
(b) RR conflict: $A \to \alpha\cdot, a \mid t_1$; $B \to \gamma\cdot, a \mid t_2$

(*iv*) Every LALR(1) grammar is CLR(1) but every CLR(1) grammar need not be LALR(1).

(*v*) LALR(1) parsers are often used in practice as parsing tables are smaller then CLR(1) parsing.

Parsers Comparision

- OPP < LL(1) < LR(0) < SLR(1) ≤ LALR(1) ≤ CLR(1)
- Number of states (SLR(1)) = Number of states (LALR(1)) ≤ Number of states (CLR(1))



- LR(0) ⊂ SLR(1) ⊂ LALR(1) ⊂ CLR(1)
- LL(1) ⊂ LALR(1) ⊂ CLR(1)

■■■

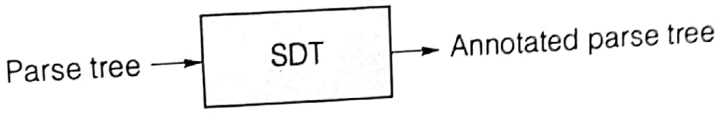
Syntax Directed Translation

3

Syntax Directed Translation (SDT)

- The CFG with semantic actions is called as SDT or the translation rules placed in RHS side of production.

Example: $S \rightarrow AB$ {print (*)}
$A \rightarrow a$ {print (1)}
$B \rightarrow S$ {print (2)}



- SDT can be constructed
  (*i*) to store the information in symbol table
  (*ii*) to verify the variable declaration
  (*iii*) to construct DAG
  (*iv*) to perform type checking, type conversion, type evaluation.
  (*v*) to evaluate algebraic expression etc.
- A parse tree which shows attributes at each and every node is called as ANNOTATED/DECORATED parse tree.

Types of Attributes

1. **Synthesized attribute:** Attribute whose value is evaluated in terms of attribute values of it's children.
2. **Inherited attribute:** Attribute whose value is evaluated in terms of attribute values of siblings or parent.

Syntax Directed Definition (SDD)

1. **S-attributed SDD:**
   (*i*) Uses "only synthesized attributes"
   (*ii*) Semantic actions can be placed at right end of the rule
   (*iii*) Follows bottom up approach
   $A \rightarrow BC \{A \cdot i = f(B \cdot i \text{ or } C \cdot i)\}$
2. **L-attributed SDD:**
   (*i*) SDT in which attributes are synthesized or restricted to inherit either from 'parent' or 'left siblings',
   (*ii*) Follows DFS and top down approach

Example: $A \rightarrow BCD$ $\{C \cdot i = f(A \cdot i \text{ or } B \cdot i)\}$ or
$\{D \cdot i = f(A \cdot i \text{ or } B \cdot i \text{ or } C \cdot i)\}$ or
$\{B \cdot i = f(A \cdot i)\}$

- Every *S*-attributed SDT is L-attributed SDT.
- Every *L*-attributed SDT need not be S-attributed SDT .
- SDT follows depth first search notation.

■■■

Runtime Environments

4

Run-time Storage Organisation

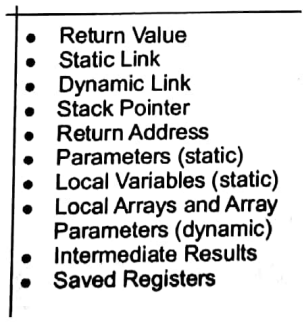
The runtime environment is the structure of the target computers registers and memory that serves to manage memory and maintain information needed to guide a programs execution process.

Types of Runtime Environments

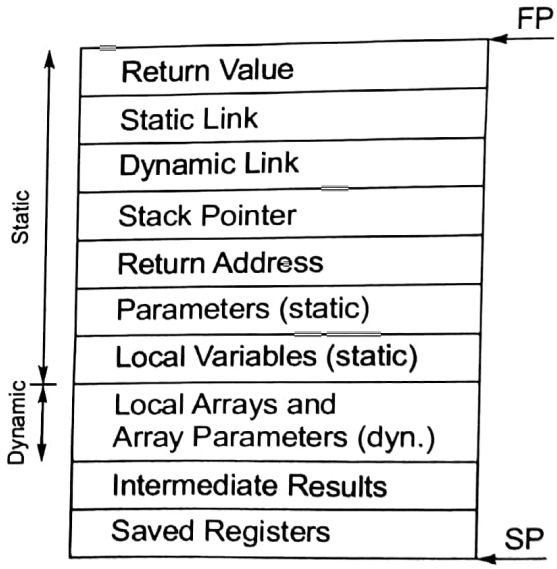
1. **Fully Static Runtime Environment:** Fully static runtime environment may be useful for the languages in which pointers or dynamic allocation is not possible in addition to no support for recursive function calls
   (*i*) Every procedure will have only one activation record which is allocated before execution.
   (*ii*) Variables are accessed directly via fixed addresses.
   (*iii*) Little book keeping overhead; i.e., at most return address may have to be stored in activation record.
   (*iv*) The calling sequence involves calculation of each argument address and storing into its appropriate parameter location and saving the return address and then a jump is made.
2. **Stack based Runtime Environment:** In this, activation records are allocated (push of the activation records) whenever a function call is made. The neccessary memory is taken from stack portion of the program. When program execution returns from function the memory used by the activation record is deallocated (pop of the activation record). Thus, stack grows and shrinks with the chain of function calls.
3. **Fully Dynamic Runtime Environment:** Functional languages such as Lisp, ML, etc. uses this style of call stack management. Sailently, here activation records are de-allocated only when all references to them have disappeared, and this requires that activation records to dynamically freed at arbitrary times during execution. Memory manager (garbage collector) is needed.
   The data structure that handles such a management is heap and thus this is also called as heap management.

Activation Records

- Information needed by a single execution of a procedure is managed using a contiguous block of storage called "activation record".
- An activation record is allocated when a procedure is entered and it is deallocated when that procedure is exited.
- It contains temporary data, local data, machine status, optional access link, optional control link, actual parameters and returned value.
  - (*i*) Program Counter (PC) whose value is the address of the next instruction to be executed.
  - (*ii*) Stack pointer (SP) whose value is top of the (top of stack, tos).
  - (*iii*) Frame Pointer (FP) which points to the current activation record.



Activation Record Contents



Activation Record

Note:

- Activation records are allocated from one of static area (like Fortran 77), stack area (like C or Pascal) and heap area (like lisp).

■■■

Intermediate, Target Code Generation and Code Optimization

5

Intermediate Code

- Intermediate codes are machine independent codes, but they are close to machine instructions.
- Syntax trees, Postfix notation, 3-address codes, DAG can be used as intermediate language
- **Three-address code:**
  - (*i*) Quadruples (4 fields: Operator, Operand1, Operand2, Result)
  - (*ii*) Triples (3 fields: Operator, Operand1, Operand2)
  - (*iii*) Indirect triples.

Need for Intermediate Code Generation

1. **Suppose we have n-source languages and m-target languages:**
   - (*i*) **Without Intermediate Code,** we will change each source language into target language directly. So, for each source-target pair we will need a compiler. Hence, we need (n × m) compliers.
   - (*ii*) **With Intermediate Code,** we need n-compilers to convert each source language into intermediate code and m-compilers to convert intermediate code into m-target languages. Thus, we need only (n + m) compilers.
2. Re-targeting is facilitated; a compiler for a different machine can be created by attaching a back-end (which generate target code) for the new machine to an existing front-end (which generate intermediate code).
3. A machine independent code-optimizer can be applied to the intermediate code.
4. Intermediate code is simple enough to be easily converted to any target code.
5. Complex enough to represent all the complex structure of high level languages.

Code Generation

A code generator has three primary tasks: instruction selection, register allocation and assignment, and instruction ordering.

1. The code generator is given the intermediate text in any one of its form.
2. But in specific algorithms, we deal with quadruples or in some case parse trees as the intermediate forms.
3. Necessary semantic checking is done.
4. The data areas and offsets have been determined for each name that information is available from the symbol table.

Code Optimization

- Optimizations applied on target code (assembly code) is machine dependent optimization.
- Optimization applied on 3-address code is machine independent optimization.
- **Basic Block:** Sequence of consecutive statements in which context enters at beginning and leaves at end without halt or possibility of branching except at the end.
- The optimization which is performed within the block is local optimization.

Machine Independent Optimizations

1. **Loop Optimization:** Process of optimizing within the loop.
   Characteristics of loop optimization:
   (*i*) Code motion/Loop invariant elimination/Frequency reduction:
   Reduce the evaluation frequency of expressions.
   Bring loop-invariant statements out of the loop
   (*ii*) **Loop Unrolling:** To execute less number of iterations.
   (*iii*) **Loop Jamming/Loop Fusion/Loop Combining:** Combine the bodies of two loops whenever they are sharing same index variable.
2. **Constant Folding:** Replacing the value of constant during compilation is called as folding.
3. **Constant propagation:** Replacing the value of an expression during compile time is known as constant propagation.

4. **Copy Propagation:**

   Example:

   $x = y \quad \Rightarrow \quad x = y$

   $z = 1 + x \qquad z = 1 + y$

5. **Dead Code Elimination:** Removes instructions without changing the behaviour (using DAG)

6. Common Subexpression Elimination/Redundant Code Elimination:

   Example:

   $x = (a + b) - (a + b)/4 \quad \Rightarrow \quad c = a + b$

   $x = c - c/4$

7. **Induction Variables and Strength Reduction:**

   An induction variable is used in loop for the following kind of assignment

   $i = i +$ constant

   Strength reduction means replacing costly operator by simple (cheap/low strength) operator.

   Example: $y = 2 * x \Rightarrow y = x+x$

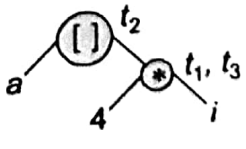
Directed Acyclic Graph (DAG)

- DAG is used to eliminate the common subexpression
- **Procedure to construct DAG:** Whenever we are going to create a node, first verify that whether node is already created or not. If it is already created then make use of the existing one instead of going for new creation.

Example:

$t_1 = 4*i$
$t_2 = a[t_1]$
$t_3 = 4*i$

$\Rightarrow$



■■■