COMPILER DESIGN

Subject Code: 10CS63 ***I.A. Marks : 25***
Hours/Week : 04 ***Exam Hours: 03***
Total Hours : 52 ***Exam Marks: 100***

PART – A

UNIT – 1 8 Hours

Introduction, Lexical analysis: Language processors; The structure of a Compiler; The evolution pf programming languages; The science of building a Compiler; Applications of compiler technology; Programming language basics.

Lexical analysis: The Role of Lexical Analyzer; Input Buffering; Specifications of Tokens; Recognition of Tokens.

UNIT – 2 6 Hours

Syntax Analysis – 1: Introduction; Context-free Grammars; Writing a Grammar. Top-down Parsing; Bottom-up Parsing.

UNIT – 3 6 Hours

Syntax Analysis – 2: Top-down Parsing; Bottom-up Parsing.

UNIT – 4 6 Hours

Syntax Analysis – 3: Introduction to LR Parsing: Simple LR; More powerful LR parsers (excluding Efficient construction and compaction of parsing tables) ; Using ambiguous grammars; Parser Generators.

PART – B

UNIT – 5 7 Hours

Syntax-Directed Translation: Syntax-directed definitions; Evaluation orders for SDDs; Applications of syntax-directed translation; Syntax-directed translation schemes.

UNIT – 6 6 Hours

Intermediate Code Generation: Variants of syntax trees; Three-address code; Translation of expressions; Control flow; Back patching; Switch statements; Procedure calls.

UNIT – 7 6 Hours

Run-Time Environments: Storage Organization; Stack allocation of space; Access to non-local data on the stack; Heap management; Introduction to garbage collection.

UNIT – 8 7 Hours

Code Generation: Issues in the design of Code Generator; The Target Language; Addresses in the target code; Basic blocks and Flow graphs; Optimization of basic blocks; A Simple Code Generator

Text Books:

1. Alfred V Aho, Monica S.Lam, Ravi Sethi, Jeffrey D Ullman: Compilers- Principles, Techniques and Tools, 2nd Edition, Pearson Education, 2007.

(Chapters 1, 3.1 to 3.4, 4 excluding 4.7.5 and 4.7.6, 5.1 to 5.4, 6.1, 6.2, 6.4, 6.6, 6.7 to 6.9, 7.1 to 7.5, 8.1 to 8.6.)

Reference Books:

1. Charles N. Fischer, Richard J. leBlanc, Jr.: Crafting a Compiler with C, Pearson Education, 1991.
2. Andrew W Apple: Modern Compiler Implementation in C, Cambridge University Press, 1997.
3. Kenneth C Louden: Compiler Construction Principles & Practice, Cengage Learning, 1997.

INDEX SHEET

COMPILER DESIGN
SUBJECT CODE: 10CS63

Preliminaries Required

- Basic knowledge of programming languages.
- Basic knowledge of DFA and NFA(FAFL concepts).
- Knowledge of a high programming language for the programming assignments.

Textbook:

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Monica,
"*Compilers: Principles, Techniques, and Tools*"
Addison-Wesley, 2nd Edition.

Course Outline

- Introduction to Compiling
- Lexical Analysis
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Semantic Analysis, Type Checking
- Run-Time Organization
- Intermediate Code Generation

UNIT I: INTRODUCTION, LEXICAL ANALYSIS

SYLLABUS:

- Lexical analysis: Language processors;
- The structure of a Compiler;
- The evolution of programming languages;
- The science of building a Compiler;
- Applications of compiler technology;
- Programming language basics.
- **Lexical analysis:** The Role of Lexical Analyzer;
- Input Buffering;
- Specifications of Tokens;
- Recognition of Tokens.

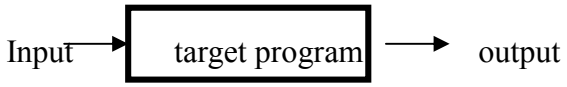
LANGUAGE PROCESSORS?

- Is a Software which process a program given in a certain source language
- Types:- compilers, Interpreters, Preprocessors, assembler ..etc

COMPILER

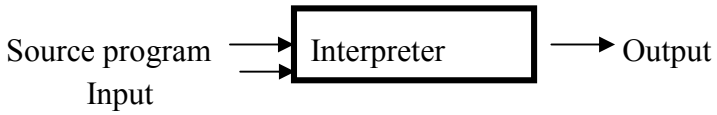
Pictorial representation of compiler is given below

If target program is executable machine language then

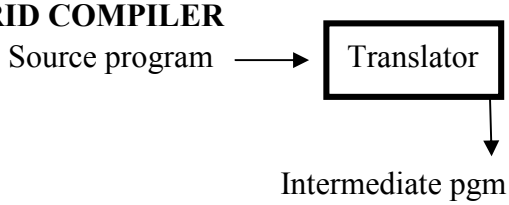


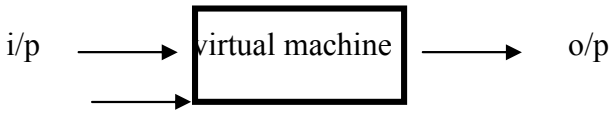
INTERPRETER

Pictorial representation of an Interpreter is given below



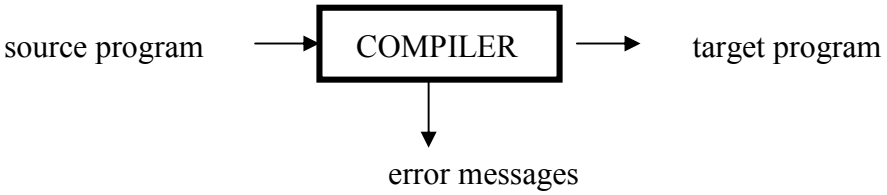
HYBRID COMPILER





COMPILERS

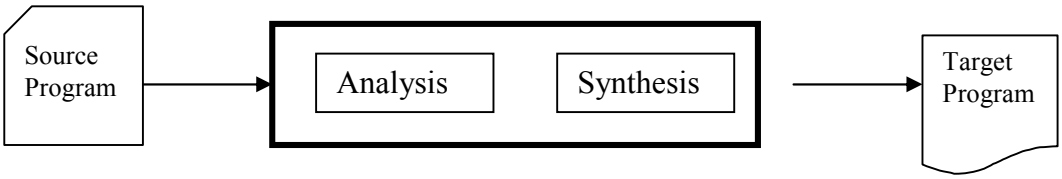
- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



Other Applications

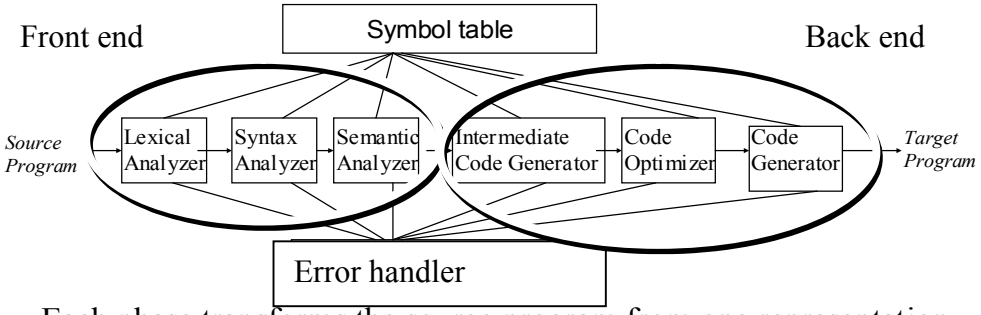
- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL.
  - Many software having a complex front-end may need techniques used in compiler design.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

Major Parts of Compilers



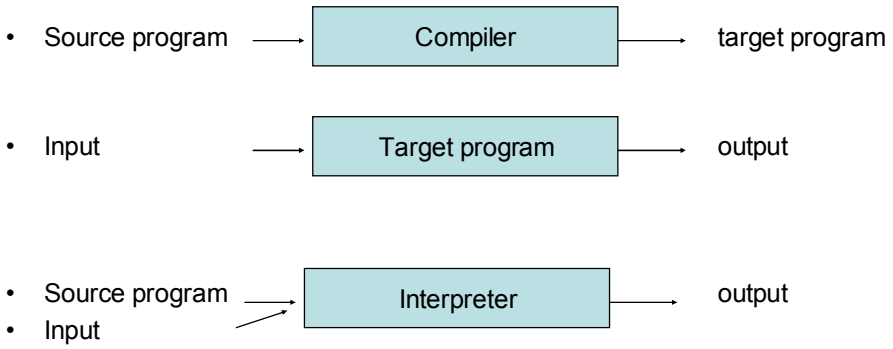
- There are two major parts of a compiler: **Analysis** and **Synthesis**
- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Phases of A Compiler

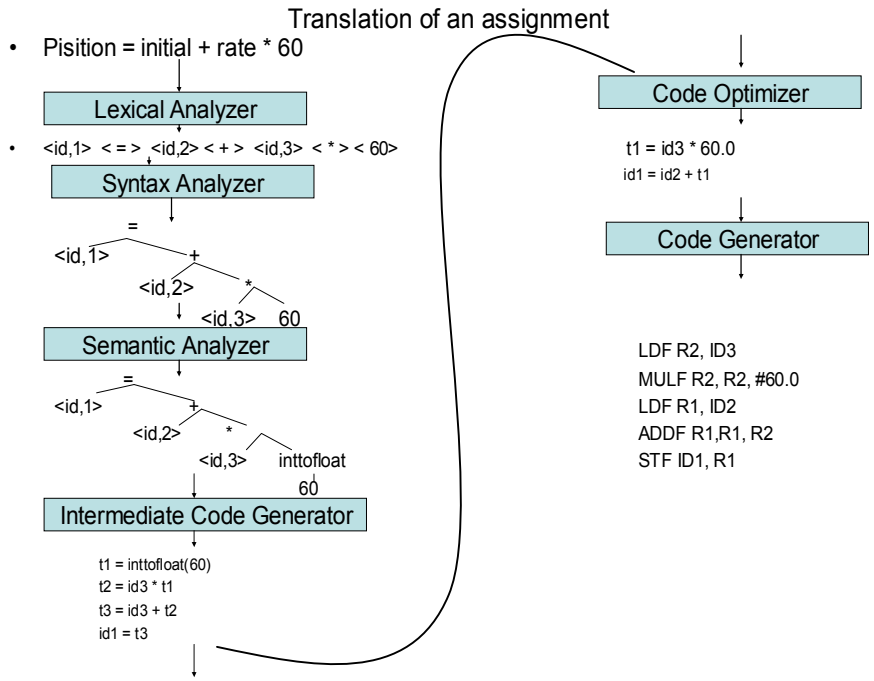


- Each phase transforms the source program from one representation into another representation.
- They communicate with error handlers.
- They communicate with the symbol table.

Compiler v/s interpreter



- *Assignments:*
- *1. What is the difference between a compiler and an interpreter?*
- *2. what are advantages of (i) a compiler over an interpreter (ii) an interpreter over a compiler?*



Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- Lexical Analyzer also called as Scanner.
- It reads the stream of characters making up the source program and groups the characters into meaningful sequences called **Lexemes.**
- For each lexeme, it produces as output a token of the form
  - *<token_name, attribute_value>*
  - *token_name* is an abstract symbol that is used during syntax analysis, and the second component *attribute_value* points to an entry in the symbol table for this token.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

  Ex: newval := oldval + 12 => tokens:

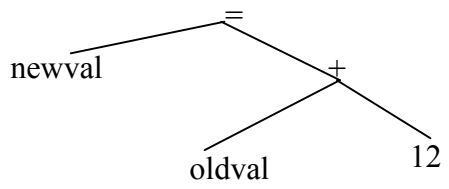
newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

- Puts information about identifiers into the symbol table not all attributes.
- Regular expressions are used to describe tokens (lexical constructs).

- A (Deterministic) Finite State Automaton(DFA) can be used in the implementation of a lexical analyzer.
- Blank spaces removed

Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar
- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
    1. If it satisfies, the syntax analyzer creates a parse tree for the given program.
- Ex: We use BNF (Backus Naur Form) to specify a CFG
    1. E →Identifier
    2. E→Number
    3. E→E+E
    4. E→E*E
    5. E→(E)
        - Where E is an expression

Syntax Analyzer list out the followings
By rule 1, newval is an expresiion
By rule 1, oldval is an expression
By rule 2, 12 is an expression
By rule 3, we get oldval+12 is an expression

Syntax Analyzer versus Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
    - Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
    - The syntax analyzer deals with recursive constructs of the language.

- The lexical analyzer simplifies the job of the syntax analyzer.
- The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
- The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - *Top-Down Parsing, Bottom-Up Parsing*
- **Top-Down Parsing:**
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).
- **Bottom-Up Parsing:**
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-Precedence Parsing – simple, restrictive, easy to implement
  - LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Determines meaning of source string.
  - Matching of parenthesis
  - Matching if else stmt.
  - Checking scope of operation
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars
- Ex:

newval = oldval + 12

- The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Intermediate Code Generation

- A compiler may produce an explicit intermediate codes representing the source program.
- Is a kind of code.
- Easy to generate
- Easily converted to target code
- It can be in three address code or quadruples, triples etc
- Ex:

newval = oldval + 1

↓

id1 = id2 + 1

↓

Intermediates Codes (Quadraples)

t1 = id2 + 1

id1= t1

3- address code

t1 = 12

t2 = id2 + 1

id1 = t2

Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.
  - T1=id2*id3
  - Id1=t1+1

Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.
- If target code is machine code then registers (memory locations) are selected for each variable in pgm. Then intermediate instructions are translated to sequences of machine instruction which perform same task.
- Ex: (assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
MOVEid2,R1
MULT id3,R1
ADD   #1,R1
MOVER1,id1
```

Symbol Table Management

- This record variable name used in the source program
- This stores attributes of each name
    - Eg: name, its type, its scope
    - Method of passing each argument(by value or by reference)
    - Return type

Implementation of symbol table can be done is either linear list or hash table. There will be ***n*** entries ***e*** enquires to fetch information from this table.

If ***n*** and ***e*** are more than linear list method is poor in performance. But hash table is better than list method

Grouping of phases into passes

- In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file.
- For example,
    - front-end phases of lexical analysis, syntax analysis, semantic analysis and from back end intermediate code generation might be grouped together into one pass.
    - Code optimization might be an optional pass.
    - The there could be a back-end pass consisting of code generation for a particular target machine.

CLASSIFICATION OF LANGUAGES

1.Based on generation:

a)First generation language-machine languages.

b)second generation languages-assembly languages.

c)Third generation languages-higher level languages.

d)Fourth generation languages-designed for specific applications like sql for database applications.

e)Fifth generation languages-applied to logic and constraint based languages like prolog and ops5.

Imperative languages:

languages in which a program specifies how a computation is to be done.

eg: c, c++.

Declarative languages:

languages in which a program specifies what computation is to be done.

eg: prolog.

The science of building a compiler

- Compiler design deals with complicated real world problems.
- First the problem is taken.
- A mathematical abstraction is formulated.
- Solve using mathematical techniques.

Modeling in compiler design and implementation

- Right mathematical model and right algorithm.
- Fundamental models – finite state machine, regular expression, context free grammar.

The science of code optimization

Optimization : It is an attempt made by compiler to produce code that is more efficient than the obvious one.

Compiler optimizations-Design objectives

- Must improve performance of many programs.
- Optimization must be correct.
- Compilation time must be kept reasonable.
- Engineering effort required must be manageable.

APPLICATIONS OF COMPILER TECHNOLOGY

IMPLEMENTATION OF HIGH LEVEL PROGRAMING LANGUAGES.

- The programmer expresses an alg using the Lang, and the compiler must translate that prog to the target language.
- Generally HLP langs are easier to program in, but are less efficient, i.e., the target prog runs more slowly.
- Programmers using LLPL have more control over a computation and can produce more efficient code.
- Unfortunately, LLP are harder to write and still worse less portable, more prone to errors and harder to maintain.
- Optimizing compilers include techniques to improve the performance of general code, thus offsetting the inefficiency introduced by HL abstractions.

OPTIMIZATIONS FOR COMPUTER ARCHITECTURES

- The rapid evolution of comp architecture has also led to an insatiable demand for a new complier technology.
- Almost all high performance systems take advantage of the same basic 2 techniques: parallelism and memory hierarchies.

- Parallelism can be found at several levels : at the instruction level, where multiple operations are executed simultaneously and at the processor level, where different threads of same application are run on different processors.
- Memory hierarchies are a response to the basic limitation that we can built very fast storage or very large storage, but not storage that is both fast and large.

PARALLELISM

- All modern microprocessors exploit instruction-level parallelism. this can be hidden from the programmer.
- The hardware scheduler dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible.
- Whether the hardware reorders the instruction or not, compilers can rearrange the instruction to make instr-level parallelism more effective.

MEMORY HIERARCHIES.

- a memory hierarchy consists of several levels of storage with different speeds and sizes.
- a processor usually has a small number of registers consisting of hundred of bytes, several levels of caches containing kilobytes to megabytes, and finally secondary storage that contains gigabytes and beyond.
- Correspondingly, the speed of accesses between adjacent levels of the hierarchy can differ by two or three orders of magnitude.
- The performance of a system is often limited not by the speed of the processor but by the performance of the memory subsystem.
- While compliers traditionally focus on optimizing the processor execution, more emphasis is now placed on making the memory hierarchy more effective.

DESIGN OF NEW COMPUTER ARCHITECTURES.

- In modern computer arch development, compilers are developed in the processor-design stage, and compiled code running on simulators, is used to evaluate the proposed architectural design.
- One of the best known ex of how compilers influenced the design of computer arch was the invention of RISC (reduced inst-set comp) arch.
- Over the last 3 decades, many architectural concepts have been proposed. they include data flow machines, vector machines, VLIW(very long inst word) machines, multiprocessors with shared memory, and with distributed memory.
- The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology.
- Compiler technology is not only needed to support programming of these arch, but also to evaluate the proposed architectural designs.

PROGRAM TRANSLATIONS

Normally we think of compiling as a translation of a high level Lang to machine level Lang, the same technology can be applied to translate between diff kinds of languages.

The following are some of the imp applications of program translation techniques:

BINARY TRANSLATION

- Compiler technology can be used to translate the binary code for one machine to that of another, allowing a machine to run programs originally compiled for another instr set.
- This tech has been used by various computer companies to increase the availability of software to their machines.

HARDWARE SYNTHESIS

- Not only is most software written in high level languages, even hardware designs are mostly described in high level hardware description languages like verilog and VHDL(very high speed integrated circuit hardware description languages).
- Hardware designs are typically described at the register transfer level (RTL).
- Hardware synthesis tools translate RTL descriptions automatically into gates which are then mapped to transistors and eventually to a physical layout. This process takes long hours to optimize the circuits unlike compilers for programming langs.

DATABASE QUERY INTERPRETERS

- Query languages like SQL are used to search databases.
- These database queries consist of relational and Boolean operators.
- They can be compiled into commands to search a database for records satisfying the needs.

SOFTWARE PRODUCTIVITY TOOLS

Several ways in which program analysis, building techniques originally developed to optimize code in compilers, have improved software productivity.

TYPE CHECKING
BOUNDS CHECKING
MEMORY MANAGEMENT TOOLS.

Programming Language Basics

Static/Dynamic Distinction

- The language uses a policy that allows the compiler to decide an issue then that language uses staticor issue decided at compile time.
- The decision is made during execution of a program is said to be dynamic or decision at run time.
- Scope of declarations.
- Eg:public static int x;

Environments and States

Programming language changes occuring as the program runs affect the values of data elements.eg:x=y+1;
Names locations values.

```
Eg:   …..
       Int i;
       …..
      Void f(….)
      {
         int I;
         …..
         i=3;
       }
            x=i+1;
```

Static Scope and Block Structure

Most of the languages,including C and its family uses static scope.The scope rules for C are based on prgm struc,the scope of declaration is determined implicitly where the declaration appears in the prgm…for java,c++ provide explicit control over scopes by using the keywords like PUBLIC,PRIVATE,and PROTECTED

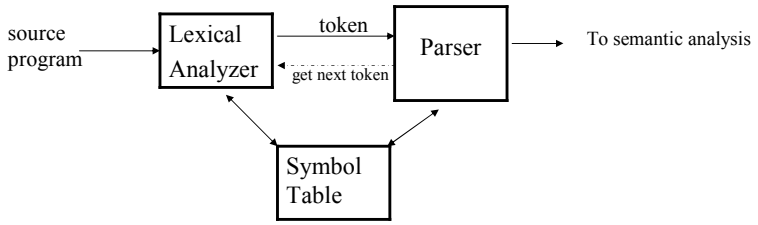
Static_scope rules for a language with blocks-grouping of declarations and statements.. e.g.: C uses braces

```
Main()
{
  int a=1;
  int b=1;
  {
     int b=2;
  {
    int a=3;
   Cout<< a< <b;
   }
  {
    int b=4;
    Cout << a<< b;
   }
Cout << a<< b;
  }
Cout << a<< b
};
```

Lexical Analyzer

THE ROLL OF THE LEXICAL ANALYZER

- **Lexical Analyzer** reads the source program character by character to produce tokens.
- Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.



Some Other Issues in Lexical Analyzer

- Skipping comments (stripping out comments & white space)
  - Normally L.A. don't return a comment as a token.
  - It skips a comment, and return the next token (which is not a comment) to the parser.
  - So, the comments are only processed by the lexical analyzer, and the don't complicate the syntax of the language.
- Correlating error messages

  - It can associate a line number with each error message..

  - In some compilers it makes a copy of the source pgm. with the error messages inserted at the appropriate positions.

  - If the source pgm. Using **macro-processor**, the expansion of macros may be performed by the l.a
- Symbol table interface
  - symbol table holds information about tokens (at least lexeme of identifiers)
  - how to implement the symbol table, and what kind of operations.
    - **hash table – open addressing, chaining**
    - **putting into the hash table, finding the position of a token from its lexeme.**

Token: It describes the class or category of input string. For example, identifier, keywords, constants are called tokens

Patters: set of rule that describes the tokens

Lexeme: sequence of character in the source pgm that are matched with the pattern of the token. Eg: int, I, num, ans etc

Implementing Lexical Analyzers

- Different approaches:
  Using a scanner generator, e.g., lex or flex. This automatically generates a lexical analyzer from a high-level description of the tokens. (easiest to implement; least efficient)
- Programming it in a language such as C, using the I/O facilities of the language. (intermediate in ease, efficiency)
- Writing it in assembly language and explicitly managing the input. (hardest to implement, but most efficient)

List out lexeme and token in the following example

1.

```
int Max(int a, int b)
{ if(a>b)
     return a;
  else
     return b;
}
```

Lexeme	Token
int	Keyword
Max	Identifier
(	Operator
a	Identifier
,	Operator
.	.
.	.
.	.

2. **Examples of tokens**

Token	**Informal description**	**Sample Lexemes**
if	Characters i, f	if
else	Characters e, l, s, e	else
Comparison	< or> or <= or >= or == or !=	<=, !=
id	Letter followed by letters and digits	Pi, score, D2
Number	Any numeric constant	3.14, 0.6, 20
Literal	Anything but surrounded by " " 's	"total= %d\n" , "core dumped"

In FORTRAN,

DO 5 I = 1.25 → DO5I is a lexeme

DO 5 I = 1, 25 → do- statement

Disadvantage of Lexical Analyzer

Another disadvantage of LA is

Instead of if(a==b) statement if we mistype it as fi(a==b) then lexical analyzer will not rectify this mistake.

Advantages of Lexical Analyzer

Lexical analysis v/s Parsing

- Simplicity of design is the most important consideration.
- Compiler efficiency is improved.
  - can apply specialized techniques that serve only the lexical task
  - Specialized buffering techniques for reading input characters can speed up the compiler.
- Compiler portability is enhanced.
  - Input-device-specific peculiarities can be restricted to the lexical analyzer.

Lexical analysis v/s Parsing

- Simplicity of design is the most important consideration.
- Compiler efficiency is improved.
  - can apply specialized techniques that serve only the lexical task
  - Specialized buffering techniques for reading input characters can speed up the compiler.
- Compiler portability is enhanced.
  - Input-device-specific peculiarities can be restricted to the lexical analyzer.

Input Buffering

To recognize tokens reading data/ source program from hard disk is done. Accessing hard disk each time is time consuming so special buffer technique have been developed to reduce the amount of overhead required.

- One such technique is two-buffer scheme each of which is alternately loaded.
- Size of each buffer is N(size of disk block) Ex:4096 bytes
- One read command is used to read N characters.
- If fewer than N characters remain in the input file , then special character, represented by **eof**, marks the end of source file.

.**Sentinel** is a special character that cannot be a part of source program. eof is used as sentinel

- Two pointers to the input are maintained
  - Pointer *lexemeBegin,* marks the beginning of the current lexeme, whose extent we are attempting to determine.
  - Pointer *forward* scans ahead until a pattern match if found.

Algorithm for i/p buffer (or)

Lookahead code with sentinels

```
Switch(*forward++)
  {
  case eof:
            if (forward is at end of first buffer)
            {  reload second buffer;
               forward = beginning of second buffer;
            }
      else if (forward is at end of second buffer)
            {  reload first buffer;
               forward = beginning of first buffer;
            }
      else /* eof within a buffer marks the end of input terminate lexical analysis*/
      break;
       case for the other characters
  }
```

15

Terminology of Languages

- **Alphabet** : a finite set of symbols (ASCII characters)
- **String** :
  - Finite sequence of symbols on an alphabet
  - Sentence and word are also used in terms of string
  - ε is the empty string
  - |s| is the length of string s.
- **Language**: sets of strings over some fixed alphabet
  - ∅ the empty set is a language.
  - {ε} the set containing empty string is a language
  - The set of well-wormed C programs is a language
  - The set of all possible identifiers is a language.
- **Operators on Strings**:
  - *Concatenation*: xy represents the concatenation of strings x and y. s ε = s ε s = s
  - s^n = s s s .. s (n times) s^0 = ε

Operations on Languages

- Concatenation:
  - $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
- Union
  - $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$
- Exponentiation:
  - $L^0 = \{\varepsilon\}$ $L^1 = L$ $L^2 = LL$
- Kleene Closure
  - $L^* =$
- Positive Closure
  - $L^+ =$

Example

- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$
- $L_1L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$
- L_1^3 = all strings with length three (using a,b,c,d}
- L_1^* = all strings using letters a,b,c,d and empty string
- L_1^+ = doesn't include the empty string

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\,(r_2)$	$L(r_1)\,L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

Regular Expressions (cont.)

- We may remove parentheses by using precedence rules.
  - * highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$

- Ex:
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon,0,00,000,0000,....\}$
  - $(0|1)^* \Rightarrow$ all strings with 0 and 1, including the empty string

Regular Definitions

- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

. $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

$d_n \rightarrow r_n$

basic symbols previously defined names

Regular Definitions

- Ex: Identifiers in Pascal

  letter → A | B | ... | Z | a | b | ... | z

  digit → 0 | 1 | ... | 9

  id → letter (letter | digit) *

  – If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

  (A|...|Z|a|...|z) ((A|...|Z|a|...|z) | (0|...|9)) *

- Ex: Unsigned numbers in Pascal

  digit → 0 | 1 | ... | 9

  digits → digit $^+$

  opt-fraction → (. digits) ?

  opt-exponent → (E (+|-)? digits) ?

  unsigned-num → digits opt-fraction opt-exponent

Recognition of Tokens

- Consider the following grammar for branching statement

stmt → if *expr* then *stmt*
| if *expr* then *stmt* else *stmt*
| €

expr → *term* relop *term*
| *term*

term → id
| number

The terminals of the grammar are ' if, then, else, relop, id and number ' ,which are the names of tokens for the lexical analyzer.

Recognition of Tokens

Our current goal is to perform the lexical analysis needed for the following grammar.

stmt → if expr then stmt
| if expr then stmt else stmt
| ε
expr → term relop term // relop is relational operator =, >, etc
| term
term → id
| number

Recall that the terminals are the tokens, the nonterminals produce terminals.

A regular definition for the terminals is

digit → [0-9]
digits → digits$^+$
number → digits (. digits)? (E[+-]? digits)?
letter → [A-Za-z]
id → letter (letter | digit)*
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>

Lexeme	Token	Attribute
Whitespace	ws	—
if	if	—
then	then	—
else	else	—
An identifier	id	Pointer to table entry
A number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

On the board show how this can be done with just REs.

We also want the lexer to remove whitespace so we define a new token

ws → (blank | tab | newline) +

where blank, tab, and newline are symbols used to represent the corresponding ascii characters.

Recall that the lexer will be called by the parser when the latter needs a new token. If the lexer then recognizes the token ws, it does *not* return it to the parser but instead goes on to recognize the next token, which is then returned. Note that you can't have two consecutive ws tokens in the input because, for a given token, the lexer will match the **longest** lexeme starting at the current position that yields this token. The table on the right summarizes the situation.

For the parser, all the relational ops are to be treated the same so they are all the same token, relop. Naturally, other parts of the compiler, for example the code generator, will need to distinguish between the various relational ops so that appropriate code is generated. Hence, they have distinct attribute values.

Specification of Token

To specify tokens Regular Expressions are used.

Recognition of Token

To recognize tokens there are 2 steps

1. Design of Transition Diagram
2. Implementation of Transition Diagram

Transition Diagrams

A transition diagram is similar to a flowchart for (a part of) the lexer. We draw one for each possible token. It shows the decisions that must be made based on the input seen. The two main components are circles representing *states* (think of them as decision points of the lexer) and arrows representing *edges* (think of them as the decisions made).

The transition diagram (3.13) for relop is shown on the right.

1. The double circles represent *accepting* or *final* states at which point a lexeme has been found. There is often an action to be done (e.g., returning the token), which is written to the right of the double circle.
2. If we have moved one (or more) characters too far in finding the token, one (or more) stars are drawn.
3. An imaginary start state exists and has an arrow coming from it to indicate where to begin the process.

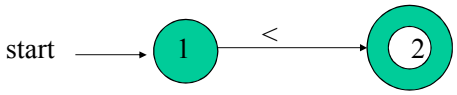
It is fairly clear how to write code corresponding to this diagram. You look at the first character, if it is <, you look at the next character. If that character is =, you return (relop,LE) to the parser. If instead that character is >, you return (relop,NE). If it is another character, return (relop,LT) and adjust the input buffer so that you will read this character again since you have not used it for the current lexeme. If the first character was =, you return (relop,EQ).

Transition Diagrams

- Transition diagrams have a collection of nodes or circles, called **States.**

- Each **state** represents a **condition** that could occur during the process of scanning the input looking for a **lexeme** that matches one of several **patterns.**

- **Edges** are directed from one state of the transition diagram to another.



25

To design transition diagram following conventions are used

- Convention about Transition diagrams

1. Accepting state or final state- indicate that a lexeme has been found. action to be taken ---return a token & attribute value to the parser
2. A * is placed at the final state, to retract the forward pointer one position.
3. Start state or initial state indicated by an edge labeled "start" , entering from nowhere.

Recognition of Reserved Words and Identifiers

The next transition diagram corresponds to the regular definition given previously.

Note again the star affixed to the final state.
Two questions remain.

1. How do we distinguish between identifiers and keywords such as then, which also match the pattern in the transition diagram?
2. What is (gettoken(), installID())?

We will continue to assume that the keywords are *reserved*, i.e., may not be used as identifiers. (What if this is not the case—as in Pl/I, which had no reserved words? Then the lexer does not distinguish between keywords and identifiers and the parser must.)

We will use the method mentioned last chapter and have the keywords installed into the identifier table prior to any invocation of the lexer. The table entry will indicate that the entry is a keyword.

installID() checks if the lexeme is already in the table. If it is not present, the lexeme is installed as an id token. In either case a pointer to the entry is returned.

gettoken() examines the lexeme and returns the token name, either id or a name corresponding to a reserved keyword.

The text also gives another method to distinguish between identifiers and keywords.

Completion of the Running Example

So far we have transition diagrams for identifiers (this diagram also handles keywords) and the relational operators. What remains are whitespace, and numbers, which are respectively the simplest and most complicated diagrams seen so far.

Recognizing Whitespace

The diagram itself is quite simple reflecting the simplicity of the corresponding regular expression.

- The delim in the diagram represents any of the whitespace characters, say space, tab, and newline.
- The final star is there because we needed to find a non-whitespace character in order to know when the whitespace ends and this character begins the next token.
- There is no action performed at the accepting state. Indeed the lexer does *not* return to the parser, but starts again from its beginning as it still must find the next token.

Recognizing Numbers

This certainly looks formidable, but it is not that bad; it follows from the regular expression.

In class go over the regular expression and show the corresponding parts in the diagram.

When an accepting states is reached, action is required but is not shown on the diagram. Just as identifiers are stored in a identifier table and a pointer is returned, there is a corresponding number table in which numbers are stored. These numbers are needed when code is generated. Depending on the source language, we may wish to indicate in the table whether this is a real or integer. A similar, but more complicated, transition diagram could be produced if the language permitted complex numbers as well.

Architecture of a Transition-Diagram-Based Lexical Analyzer

The idea is that we write a piece of code for each decision diagram. I will show the one for relational operations below. This piece of code contains a case for each state, which typically reads a character and then goes to the next case depending on the character read. The numbers in the circles are the names of the cases.

Accepting states often need to take some action and return to the parser. Many of these accepting states (the ones with stars) need to restore one character of input. This is called retract() in the code.

What should the code for a particular diagram do if at one state the character read is not one of those for which a next state has been defined? That is, what if the character read is not the label of any of the outgoing arcs? This means that we have failed to find the token corresponding to this diagram.

The code calls fail(). This is **not** an error case. It simply means that the current input does not match this particular token. So we need to go to the code section for another diagram after restoring the input pointer so that we start the next diagram at the point where this failing diagram **started**. If we have tried all the diagram, then we have a real failure and need to print an error message and perhaps try to repair the input.

Note that the order the diagrams are tried is important. If the input matches more than one token, the first one tried will be chosen.

```
TOKEN getRelop()                    // TOKEN has two components
  TOKEN retToken = new(RELOP);        // First component set here
  while (true)
    switch(state)
      case 0: c = nextChar();
              if (c == '<')      state = 1;
              else if (c == '=') state = 5;
              else if (c == '>') state = 6;
              else fail();
              break;
      case 1: ...
      ...
      case 8: retract();  // an accepting state with a star
              retToken.attribute = GT;  // second component
              return(retToken);
```

Alternate Methods

The book gives two other methods for combining the multiple transition-diagrams (in addition to the one above).

1. Unlike the method above, which tries the diagrams one at a time, the first new method tries them in parallel. That is, each character read is passed to each diagram (that hasn't already failed). Care is needed when one diagram has accepted the input, but others still haven't failed and may accept a longer prefix of the input.
2. The final possibility discussed, which appears to be promising, is to combine all the diagrams into one. That is easy for the example we have been considering because all the diagrams begin with different characters being matched. Hence we just have one large start with multiple outgoing edges. It is more difficult when there is a character that can begin more than one diagram.

UNIT II: SYNTAX ANALYSIS - 1

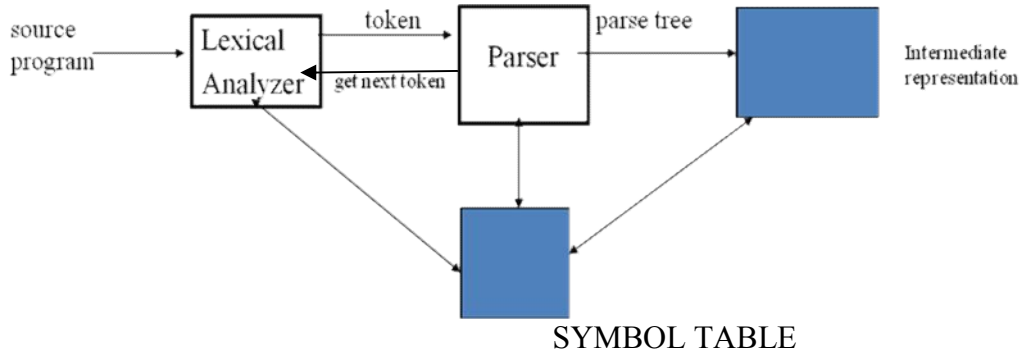
SYLLABUS

- **Syntax Analysis – 1:** Introduction;
- Context-free Grammars;
- Writing a Grammar.
- Top-down Parsing;
- Bottom-up Parsing

Introduction

- *Syntax Analyzer* creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as *parser*.
- The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - If it satisfies, the parser creates the parse tree of that program.
  - Otherwise the parser gives the error messages.
- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.
  - a grammar can be directly converted into a parser by some tools.
- Parser works on a stream of tokens.
- The smallest item is a token.

Fig :Position Of Parser in Compiler model



- We categorize the parsers into two groups:
  1. **Top-Down Parser**
  2. the parse tree is created top to bottom, starting from the root.

1. **Bottom-Up Parser**
   - the parse is created bottom to top; starting from the leaves

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

Syntax Error Handling

- Common Programming errors can occur at many different levels.

1. Lexical errors: include misspelling of identifiers, keywords, or operators.
2. Syntactic errors : include misplaced semicolons or extra or missing braces.

3. Semantic errors: include type mismatches between operators and operands.
4. Logical errors: can be anything from incorrect reasoning on the part of the programmer.

Goals of the Parser

- Report the presence of errors clearly and accurately
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

Error-Recovery Strategies

- Panic-Mode Recovery
- Phrase-Level Recovery
- Error Productions
- Global Correction

Panic-Mode Recovery

- On discovering an error, the parser discards input symbols one at a time until one of a designated set of Synchronizing tokens is found.
- Synchronizing tokens are usually delimiters.

Ex: semicolon or } whose role in the source program is clear and unambiguous.

- It often skips a considerable amount of input without checking it for additional errors.

Advantage:

Simplicity
Is guaranteed not to go into an infinite loop

Phrase-Level Recovery

- A parser may perform local correction on the remaining input. i.e

it may replace a prefix of the remaining input by some string that allows the parser to continue.

Ex: replace a comma by a semicolon, insert a missing semicolon

- Local correction is left to the compiler designer.
- It is used in several error-repairing compliers, as it can correct any input string.
- Difficulty in coping with the situations in which the actual error has occurred before the point of detection.

Error Productions

- We can augment the grammar for the language at hand with productions that generate the **erroneous constructs**.
- Then we can use the grammar augmented by these error productions to **Construct a parser.**
- If an error production is used by the parser, we can generate appropriate **error diagnostics** to indicate the erroneous construct that has been recognized in the input.

Global Correction

- We use algorithms that perform minimal sequence of changes to obtain a globally least cost correction.

- Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y.
- Such that the number of insertions, deletions and changes of tokens required to transform x into y is as small as possible.
- It is too costly to implement in terms of time space, so these techniques only of theoretical interest.

Context-Free Grammars

- Inherently recursive structures of a programming language are defined by a context-free grammar.
- In a context-free grammar, we have:
  - A finite set of terminals (in our case, this will be the set of tokens)
  - A finite set of non-terminals (syntactic-variables)
  - A finite set of productions rules in the following form
    - $A \rightarrow \alpha$ where A is a non-terminal and

      α is a string of terminals and non-terminals (including the empty string)
  - A start symbol (one of the non-terminal symbol)

NOTATIONAL CONVENTIONS

1. **Symbols used for terminals are :**
   - Lower case letters early in the alphabet (such as a, b, c, . . .)
   - Operator symbols (such as +, *, . . .)
   - Punctuation symbols (such as parenthesis, comma and so on)
   - The digits(0…9)
   - Boldface strings and keywords (such as **id** or **if**) each of which represents a single terminal symbol
2. **Symbols used for non terminals are:**
   - Uppercase letters early in the alphabet (such as A, B, C, …)
   - The letter S, which when it appears is usually the start symbol.
   - Lowercase, italic names (such as *expr* or *stmt)*.
3. **Lower case greek letters, α, β, γ for example represent (possibly empty) strings of grammar symbols.**

Example: using above notations list out terminals, non terminals and start symbol in the following example

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid id$

Here terminal are +, -, *, / , (,), id

Non terminals are E, T, F

Start symbol is E

Derivations

$E \Rightarrow E+E$

- E+E derives from E
  - we can replace E by E+E
  - to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$

- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.
- In general a derivation step is

$\alpha A\beta \Rightarrow \alpha\gamma$ if there is a production rule $A \rightarrow \gamma$ in our grammar

where α and β are arbitrary strings of terminal and non-terminal symbols

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

$\Rightarrow$: derives in one step

$\Rightarrow$: derives in zero or more steps

$\Rightarrow$: derives in one or more steps

CFG – Terminology

- L(G) is *the language of G* (the language generated by G) which is a set of sentences.
- *A sentence of L(G)* is a string of terminal symbols of G.
- If S is the start symbol of G then

ω is a sentence of L(G) iff $S \Rightarrow \omega$ where ω is a string of terminals of G.

- If G is a context-free grammar, L(G) is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.
- $S \Rightarrow \alpha$ - If α contains non-terminals, it is called as a *sentential* form of G.

  - If α does not contain non-terminals, it is called as a *sentence* of G.

Derivation Example

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

OR

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Left-Most and Right-Most Derivations

Left-Most Derivation

$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$

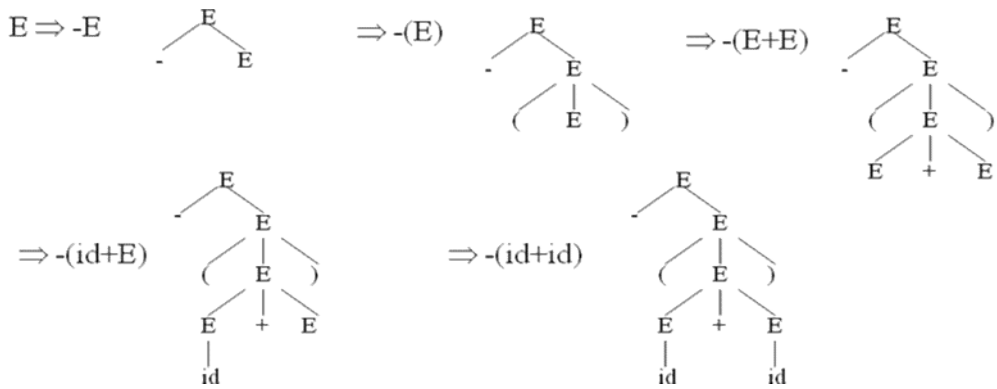
Right-Most Derivation

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$$

- We will see that the top-down parsers try to find the left-most derivation of the given source program.
- We will see that the bottom-up parsers try to find the right-most derivation of the given source program in the reverse order.

Parse Tree

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.



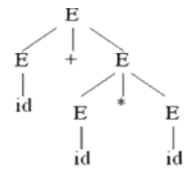
Problems on derivation of a string with parse tree:

1. Consider the grammar S→ (L) | a
   L→L,S | S
   i. What are the terminals, non terminal and the start symbol?
   ii. Find the parse tree for the following sentence
      a. (a,a)
      b. (a, (a, a))
      c. (a, ((a,a),(a,a)))
   iii. Construct LMD and RMD for each.
2. Do the above steps for the grammar S → aS | aSbS | ∈ for the string "aaabaab"

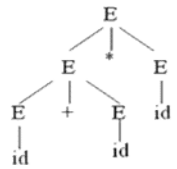
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$
$\Rightarrow id+id*E \Rightarrow id+id*id$



$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E$
$\Rightarrow id+id*E \Rightarrow id+id*id$

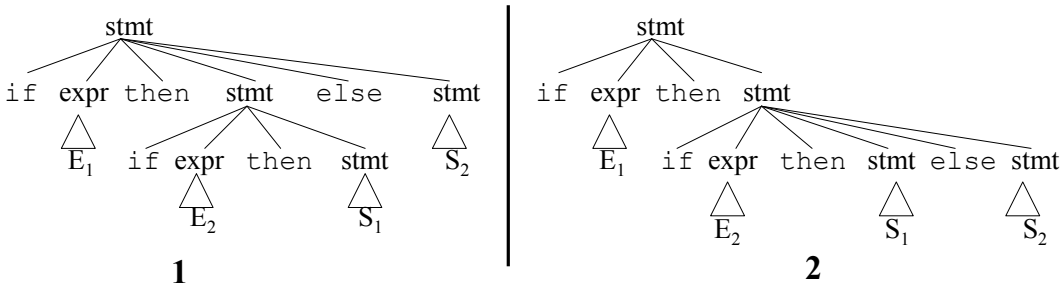


- For the most parsers, the grammar must be unambiguous.
- unambiguous grammar
  - ➔ unique selection of the parse tree for a sentence
- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An ambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.
- EG:

Ambiguity (cont.)

stmt → `if` expr `then` stmt |
`if` expr `then` stmt `else` stmt | otherstmts

`if` E_1 `then` `if` E_2 `then` S_1 `else` S_2



- We prefer the second parse tree (else matches with closest if).
- So, we have to disambiguate our grammar to reflect this choice.
- The unambiguous grammar will be:
- stmt → matchedstmt | unmatchedstmt

- matchedstmt → if expr then matchedstmt else matchedstmt | otherstmts
- unmatchedstmt → if expr then stmt |
- if expr then matchedstmt else unmatchedstmt

Problems on ambiguous grammar:

Show that the following grammars are ambiguous grammar by constructing either 2 lmd or 2 rmd for the given string.

1. S → S(S)S | ∈ with the string (( ) ( ))
2. S → S+S | |SS | (S) |S* a with the string (a+a)*a
3. S → aS | aSbS | ∈ with the string abab

Ambiguity – Operator Precedence

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

  E → E+E | E*E | E^E | id | (E)

  disambiguate the grammar

  precedence: ^ (right to left)
  * (left to right)
  + (left to right)

  E → E+T | T
  T → T*F | F
  F → G^F | G
  G → id | (E)

Left Recursion

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

  $A \overset{+}{\Rightarrow} A\alpha$ for some string α

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

⇓ eliminate immediate left recursion

$A \rightarrow \beta\, A'$

$A' \rightarrow \alpha\, A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

⇓ eliminate immediate left recursion

$A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$ an equivalent grammar

Example

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T\,E'$
$E' \rightarrow +T\,E' \mid \varepsilon$
$T \rightarrow F\,T'$
$T' \rightarrow *F\,T' \mid \varepsilon$
$F \rightarrow id \mid (E)$

Left-Recursion – Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$
$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.
$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or
$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

Eliminate Left-Recursion – Algorithm

- Arrange non-terminals in some order: $A_1 \ldots A_n$

```
- for i from 1 to n do {
    - for j from 1 to i-1 do {
        replace each production
            A_i → A_j γ
              by
            A_i → α_1 γ | ... | α_k γ
            where A_j → α_1 | ... | α_k
        }
    - eliminate immediate left-recursions among A_i productions
}
```

Example2:

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop.
- Eliminate the immediate left-recursion in A

$A \rightarrow SdA' \mid fA'$
$A' \rightarrow cA' \mid \varepsilon$

for S:

- Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid fA'a$
  So, we will have $S \rightarrow SdA'a \mid fA'a \mid b$
- Eliminate the immediate left-recursion in S

$S \rightarrow fA'aS' \mid bS'$
$S' \rightarrow dA'aS' \mid \varepsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow fA'aS' \mid bS'$
$S' \rightarrow dA'aS' \mid \varepsilon$
$A \rightarrow SdA' \mid fA'$
$A' \rightarrow cA' \mid \varepsilon$

Problems of left recursion

1. S → S(S)S | ∈
2. S → S+S | |SS | (S) |S* a
3. S → SS+ | SS* | a
4. bexpr → bexpr or bterm | bterm
   bterm →bterm and bfactor | bfactor
   bfactor → not bfactor | (bexpr) | true | false
5. S → (L) | a, L → L,S | S

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.
  grammar ➔ a new equivalent grammar suitable for predictive parsing

stmt → if expr then stmt else stmt |
if expr then stmt

- when we see if, we cannot now which production rule to choose to re-write *stmt* in the derivation.
- In general,
  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.
- when processing α we cannot know whether expand
  A to $\alpha\beta_1$ or
  A to $\alpha\beta_2$
- But, if we re-write the grammar as follows
  $A \rightarrow \alpha A'$
  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring – Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say
  $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$
  convert it into
  $A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$
  $A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$

Left-Factoring – Example1

A → <u>a</u>bB | <u>a</u>B | cdg | cdeB | cdfB
⇓

$A \rightarrow aA' \mid \underline{cdg} \mid \underline{cd}eB \mid \underline{cd}fB$
$A' \rightarrow bB \mid B$
$\Downarrow$
$A \rightarrow aA' \mid cdA''$
$A' \rightarrow bB \mid B$
$A'' \rightarrow g \mid eB \mid fB$

Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$
$\Downarrow$
$A \rightarrow aA' \mid b$
$A' \rightarrow d \mid \varepsilon \mid b \mid bc$
$\Downarrow$
$A \rightarrow aA' \mid b$
$A' \rightarrow d \mid \varepsilon \mid bA''$
$A'' \rightarrow \varepsilon \mid c$

Problems on left factor

1. S → iEtS | iEtSeS | a, E → b
2. S → S(S)S | ∈
3. S → aS | aSbS | ∈
4. S → SS+ | SS* | a
5. bexpr → bexpr or bterm | bterm
   bterm →bterm and bfactor | bfactor
   bfactor → not bfactor | (bexpr) | true | false
6. S→ 0S1 | 01
7. S → S+S | |SS | (S) |S* a
8. S → (L) | a, L → L,S | S
9. rexpr → rexpr + rterm | rterm
   rterm →rterm rfactor | rfactor
   rfactor → rfactor* | rprimary
   rprimary →a |b

do both leftfactor and left recursion

Non-Context Free Language Constructs

- There are some language constructions in the programming languages which are not context-free. This means that, we cannot write a context-free grammar for these constructions.
- L1 = { ωcω | ω is in (a | b)*} is not context-free
  - ➔ Declaring an identifier and checking whether it is declared or not later.

We cannot do this with a context-free language. We need semantic analyzer (which is not context-free).

- L2 = $\{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$ is not context-free
  - ➔ Declaring two functions (one with n parameters, the other one with m parameters), and then calling them with actual parameters.

Top – down parser

Recursive Descent parser
Predictive parser
LL(1) parser
LL(k) parser

First L stands for left to right scan
Second L stands for LMD
(1) stands for only one i/p symbol to predict the parser
(2) stands for k no. of i/p symbol to predict the parser

- The parse tree is created top to bottom.
- Top-down parser
  - Recursive-Descent Parsing
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
  - Predictive Parsing
    - no backtracking
    - efficient
    - Needs a special form of grammars (LL (1) grammars).
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.

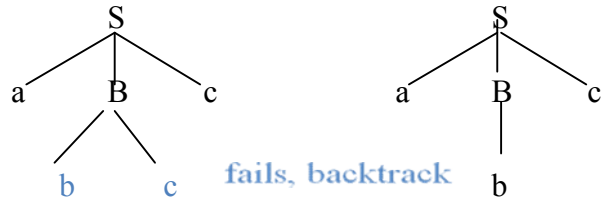
Non-Recursive (Table Driven) Predictive Parser is also known as LL (1) parser.

Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.

$S \rightarrow aBc$
$B \rightarrow bc \mid b$

input: abc



Predictive Parser

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

  $A \rightarrow \alpha_1 \mid ... \mid \alpha_n$ input: ... a .....
  current token

example

stmt → if ..... |
while ..... |
begin ..... |
for .....

- When we are trying to write the non-terminal *stmt*, if the current token is if we have to choose first production rule.

- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.

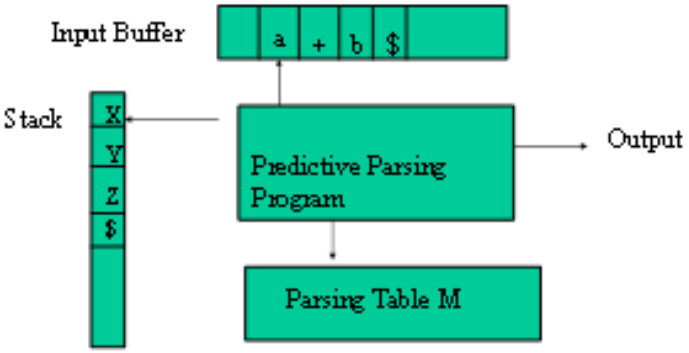


Fig: Model Of Non-Recursive predictive parsing

LL(1) Parser

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol $.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol $.
- initially the stack contains only the symbol $ and the starting symbol S. $S ← initial stack
- when the stack is emptied (ie. only $ left in the stack), the parsing is completed.

parsing table

- a two-dimensional array M[A, a]
- each row is a non-terminal symbol
- each column is a terminal symbol or the special symbol $

each entry holds a production rule.

Constructing LL(1) Parsing Tables

- Two functions are used in the construction of LL(1) parsing tables:
  - FIRST FOLLOW

- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.
- if α derives to ε, then ε is also in FIRST(α) .
- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal A* in the strings derived from the starting symbol.
  - a terminal a is in FOLLOW(A) if $S \Rightarrow \alpha A a \beta$
  - \$ is in FOLLOW(A) if $S \Rightarrow \alpha A$

Compute FIRST for Any String X

- If X is a terminal symbol ➔ FIRST(X)={X}
- If X is a non-terminal symbol and $X \rightarrow \varepsilon$ is a production rule ➔ ε is in FIRST(X).
- If X is a non-terminal symbol and $X \rightarrow Y_1Y_2..Y_n$ is a production rule

  ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1then **a** is in FIRST(X).

  ➔ if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X).
- If X is ε ➔ FIRST(X)={ε}
- If X is $Y_1Y_2..Y_n$ ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X).

  ➔ if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X).

Compute FOLLOW (for non-terminals)

- If S is the start symbol ➔ \$ is in FOLLOW(S)
- if $A \rightarrow \alpha B \beta$ is a production rule ➔ everything in FIRST(β) is FOLLOW(B) except ε
- If ($A \rightarrow \alpha B$ is a production rule) or ($A \rightarrow \alpha B \beta$ is a production rule and ε is in FIRST(β))

  ➔ everything in FOLLOW(A) is in FOLLOW(B).

We apply these rules until nothing more can be added to any follow set.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.

1. If X and a are \$ ➔ parser halts (successful completion)
2. If X and a are the same terminal symbol (different from \$)

   ➔ parser pops X from the stack, and moves the next symbol in the input buffer.
3. If X is a non-terminal

➔ parser looks at the parsing table entry M[X, a]. If M[X, a] holds a production rule $X \rightarrow Y_1Y_2...Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, ..., Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1Y_2...Y_k$ to represent a step of the derivation.

4. none of the above ➔ error
   - all empty entries in the parsing table are errors.
   - If X is a terminal symbol different from a, this is also an error case.

Non-Recursive predictive parsing Algorithm

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
    if ( X is a ) pop the stack and advance ip;
    else if ( X is a terminal ) error();
    else if ( M[X,a] is an error entry ) error();
    else if ( M[X,a] = X → Y1Y2···Yk ) {
            output the production X → Y1Y2···Yk;
            pop the stack;
            push Yk, Yk-1, ..., Y1 onto the stack, with Y1 on top;
    }
    set X to the top stack symbol;
}
```

Figure 4.20: Predictive parsing algorithm

13

LL(1) Parser – Example1

$S \rightarrow aBa$ LL (1) Parsing Table

$B \rightarrow bB \mid \varepsilon$

FIRST FUNCTION

FIRST(S) = {a} FIRST (aBa) = {a}

FIRST (B) = {b} FIRST (bB) = {b} FIRST (ε) = {ε}

FOLLOW FUNCTION

FOLLOW(S) = {\$} FOLLOW (B) = {a}

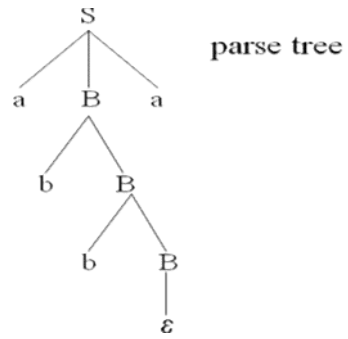
	a	**b**	**\$**
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

stack	**input**	**output**
\$S	abba\$	$S \rightarrow aBa$
\$aBa	abba\$	
\$aB	bba\$	$B \rightarrow bB$
\$aBb	bba\$	

$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



Example2

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Soln:

FIRST Example

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

FIRST(F) = {(,id}	FIRST(TE') = {(,id}	
FIRST(T') = {*, ε}	FIRST(+TE') = {+}	
FIRST(T) = {(,id}	FIRST(ε) = {ε}	
FIRST(E') = {+, ε}	FIRST(FT') = {(,id}	
FIRST(E) = {(,id}	FIRST(*FT') = {*}	
FIRST(ε) = {ε}	FIRST((E)) = {(}	FIRST(id) = {id}

FOLLOW Example

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

FOLLOW (E) = {$,)}

FOLLOW (E') = {$,)}

FOLLOW (T) = {+,), $}

FOLLOW (T') = {+,), $}
FOLLOW (F) = {+, *,), $}

Constructing LL (1) Parsing Table – Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α) ➔ add $A \rightarrow \alpha$ to M[A, a]
  - If ε in FIRST(α) ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A, a]
  - If ε in FIRST(α) and $ in FOLLOW(A) ➔ add $A \rightarrow \alpha$ to M[A, $]
  - All other undefined entries of the parsing table are error entries.

Constructing LL (1) Parsing Table – Example

$E \rightarrow TE'$ FIRST (TE') = {(, id} ➔ $E \rightarrow TE'$ into M [E, (] and M[E, id]
$E' \rightarrow +TE'$ FIRST ($+TE'$) = {+} ➔ $E' \rightarrow +TE'$ into M [E', +]
$E' \rightarrow \varepsilon$ FIRST (ε) = {ε} ➔ none
but since ε in FIRST(ε) and FOLLOW(E')={$,)}
➔ $E' \rightarrow \varepsilon$ into M[E',$] and M[$E'$,)]
$T \rightarrow FT'$ FIRST (FT') = {(, id} ➔ $T \rightarrow FT'$ into M[T,(] and M[T, id]
$T' \rightarrow *FT'$ FIRST ($*FT'$) = {*} ➔ $T' \rightarrow *FT'$ into M [T',*]
$T' \rightarrow \varepsilon$ FIRST (ε) = {ε} ➔ none
but since ε in FIRST(ε)
and FOLLOW(T')={$,) ,+}
➔ $T' \rightarrow \varepsilon$ into M [T', $], M [$T'$,)] and M [T',+]
$F \rightarrow (E)$ FIRST ((E)) = {(} ➔ $F \rightarrow (E)$ into M [F, (]
$F \rightarrow id$ FIRST (id) = {id} ➔ $F \rightarrow id$ into M [F, id]

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

stack	**input**	**output**
$E	id+id$	$E \rightarrow TE'$
$E'T	id+id$	$T \rightarrow FT'$
$E'T'F	id+id$	$F \rightarrow id$
$ E'T'id	id+id$	
$ E'T'	+id$	$T' \rightarrow \varepsilon$
$ E'	+id$	$E' \rightarrow +TE'$
$ E'T+	+id$	

$ E' T	id$	T → FT'
$ E' T' F	id$	F → id
$ E' T' id	id$	
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	accept

Construct the predictive parser LL (1) for the following grammar and parse the given string

1. S → S(S)S | ∈ with the string (( ) ( ))
2. S → + S S | |*SS | a with the string "+*aa a"
3. S → aSbS | bSaS | ∈ with the string "aabbbab"
4. bexpr → bexpr or bterm | bterm
   bterm →bterm and bfactor | bfactor
   bfactor → not bfactor | (bexpr) | true | false
   string " not(true or false)"
5. S → 0S1 | 01 string "00011"
6. S → aB | aC | Sd |Se
   B → bBc | f
   C →g
7. P → Ra | Qba
   R → aba | caba | Rbc
   Q → bbc |bc string " cababca"
8. S → PQR
   P → a | Rb | ∈
   Q → c | dP |∈
   R → e | f string " adeb"
9. E → E+ T |T
   T → id | id[ ] | id[X]
   X → E, E | E string "id[id]"
10. S → (A) | 0
    A → SB
    B → ,SB |∈ string " (0, (0,0))"
11. S → a | ↑ | (T)
    T → T,S | S
    String (a,(a,a))
    String ((a,a), ↑ , (a),a)

LL (1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL (1) grammar. one input symbol used as a look-head symbol do determine parser action LL (1) left most derivation input scanned from left to right
- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL (1) grammar.

A Grammar which is not LL (1)

S → i C t S E | a
E → e S | ε
C → b

FIRST(iCtSE) = {i}	FOLLOW(S) = {$, e}
FIRST(a) = {a}	FOLLOW (E) = {$, e}
FIRST(eS) = {e}	FOLLOW(C) = {t }
FIRST(ε) = {ε}	
FIRST(b) = {b}	

	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow e\ S$ $E \rightarrow \varepsilon$			$E \rightarrow \varepsilon$
C		$C \rightarrow b$				

two production rules for M[E, e]

Problem ➔ ambiguity

- What do we have to do it if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If it's (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL (1) grammar.
  - $A \rightarrow A\alpha \mid \beta$
    - ➔ any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.
    - ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL (1) grammar.

Properties of LL(1) Grammars

- A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$
  1. Both α and β cannot derive strings starting with same terminals.
  2. At most one of α and β can derive to ε.
  3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
  - if the terminal symbol on the top of stack does not match with the current input symbol.

  - if the top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A, a] is empty.
- What should the parser do in an error case?
  - The parser should be able to give an error message (as much as possible meaningful error message).
  - It should be recovered from that error case, and it should be able to continue the parsing with the rest of the input.

Error Recovery Techniques

- Panic-Mode Error Recovery
  - Skipping the input symbols until a synchronizing token is found.
- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- Error-Productions
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
  - When an error production is used by the parser, we can generate appropriate error diagnostics.
  - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- Global-Correction
  - Ideally, we would like a compiler to make as few changes as possible in processing incorrect inputs.
  - We have to globally analyze the input to find the error.
  - This is an expensive method, and it is not in practice.

Panic-Mode Error Recovery in LL (1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.
- What is the synchronizing token?
  - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.
- So, a simple panic-mode error recovery for the LL(1) parsing:
  - All the empty entries are marked as ***synch*** to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A which on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
  - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

Panic-Mode Error Recovery – Example

S → AbS | e | ε

A → a | cAd

Soln:

FIRST (S) = FIRST (A) = {a, c}

FIRST (A) = {a, c}

FOLLOW (S) = {$}

FOLLOW (A) = {b, d}

	a	**b**	**c**	**d**	**e**	**$**
S	**S → AbS**	***sync***	**S → AbS**	***sync***	**S → e**	**S → ε**
A	**A → a**	***sync***	**A → cAd**	***sync***	***sync***	***sync***

Eg: input string "aab"

stack	**input**	**output**
$S	aab$	S → AbS
$SbA	aab$	A → a
$Sba	aab$	
$Sb	ab$	Error: missing b, inserted
$S	ab$	S → AbS
$SbA	ab$	A → a
$Sba	ab$	
$Sb	b$	
$S	$	S → ε
$	$	accept

Eg: Another input string "ceadb"

stack	**input**	**output**
$S	ceadb$	S → AbS
$SbA	ceadb$	A → cAd
$SbdAc		ceadb$
$SbdA	eadb$	Error:unexpected e (illegal A)
(Remove all input tokens until first b or d, pop A)		
$Sbd	db$	
$Sb	b$	
$S	$	S → ε
$	$	accept

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - Change, insert, or delete input symbols.

  - issue appropriate error messages
  - Pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.

UNIT III: SYNTAX ANALYSIS-2

SYLLABUS

- Top-down Parsing;
- Bottom-up Parsing

Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root.
- A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

  $S \Rightarrow ... \Rightarrow \omega$ (the right-most derivation of ω)

  $\leftarrow$ (the bottom-up parser finds the right-most derivation in the reverse order)
- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
  - At each shift action, the current symbol in the input string is pushed to a stack.
  - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.
  - There are also two more actions: accept and error.

Shift-Reduce Parsing

- A shift-reduce parser tries to reduce the given input string into the starting symbol.

a string ➔ the starting symbol

reduced to

- At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.
- If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.

Rightmost Derivation: $S \overset{rm}{\Rightarrow} \omega$

Shift-Reduce Parser finds: $\omega \overset{rm}{\Leftarrow} ... \overset{rm}{\Leftarrow} S$

Example

$S \rightarrow aABb$	input string:	aaabb
$A \rightarrow aA \mid a$		aaAbb
$B \rightarrow bB \mid b$		aAbb $\Downarrow$ reduction
		aABb
		S

$$S \underset{rm}{\Rightarrow} aABb \underset{rm}{\Rightarrow} aAbb \underset{rm}{\Rightarrow} aaAbb \underset{rm}{\Rightarrow} aaabb$$

Right Sentential Forms

- How do we know which substring to be replaced at each reduction step?

Handle

- Informally, a **handle** of a string is a substring that matches the right side of a production rule.
  - But not every substring matches the right side of a production rule is handle
- A **handle** of a right sentential form γ ($\equiv \alpha\beta\omega$) is
  a production rule $A \rightarrow \beta$ and a position of γ
  where the string β may be found and replaced by A to produce
  the previous right-sentential form in a rightmost derivation of γ.

$$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A\omega \underset{rm}{\Rightarrow} \alpha\beta\omega$$

- If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.
- We will see that ω is a string of terminals.

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.
  $S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \omega$ ← input string
- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.
  - Handle pruning help in finding handle which will be reduced to a non terminal, that is the process of shift reduce parsing.

A Shift-Reduce Parser

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

Right-Most Derivation of id+id*id

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$
$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$

Right-Most Sentential Form	Reducing Production
id+id*id	$F \rightarrow id$
F+id*id	$T \rightarrow F$
T+id*id	$E \rightarrow T$
E+id*id	$F \rightarrow id$
E+F*id	$T \rightarrow F$
E+T*id	$F \rightarrow id$
E+T*F	$T \rightarrow T*F$
E+T	$E \rightarrow E+T$
E	

Handles are red and underlined in the right-sentential forms.

A Stack Implementation of A Shift-Reduce Parser

- There are four possible actions of a shift-parser action:
  1. **Shift** : The next input symbol is shifted onto the top of the stack.
  2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
  3. **Accept**: Successful completion of parsing.
  4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.
- Initial stack just contains only the end-marker $.
- The end of the input string is marked by the end-marker $.

Consider the following grams and parse the respective strings using shift-reduce parser.

(1) $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$ string is "id + id * id"

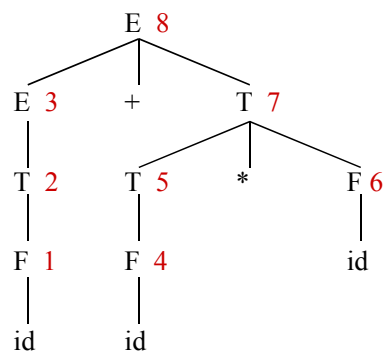
Here we follow 2 rules

1. If the incoming operator has more priority than in stack operator then perform shift.
2. If in stack operator has same or less priority than the priority of incoming operator then perform reduce.

A Stack Implementation of A Shift-Reduce Parser

Stack	**Input**	**Action**
$	id+id*id$	shift
$id	+id*id$	reduce by F → id
$F	+id*id$	reduce by T → F
$T	+id*id$	reduce by E → T
$E	+id*id$	shift
$E+	id*id$	shift
$E+id	*id$	reduce by F → id
$E+F	*id$	reduce by T → F
$E+T	*id$	shift
$E+T*	id$	shift
$E+T*id	$	reduce by F → id
$E+T*F	$	reduce by T → T*F
$E+T	$	reduce by E → E+T
$E	$	accept

Parse Tree



(2) S →TL;
T →int | float
L → L, id | id
String is "int id, id;" do shift-reduce parser.

(3) S →(L) |a
L → L,S | S
String "(a,(a,a))" do shift-reduce parser.

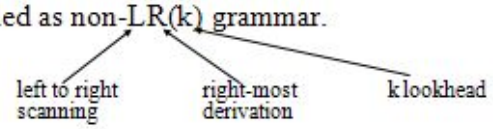
Shift reduce parser problem

- Take the grammar:
- Sentence --> NounPhrase VerbPhrase NounPhrase --> Art Noun
  VerbPhrase --> Verb | Adverb Verb Art --> the | a | ...
  Verb --> jumps | sings | ... Noun --> dog | cat | ...
  And the input: "the dog jumps". Then the bottom up parsing is:

Stack	**Input Sequence**	**ACTION**
$	the dog jumps$	SHIFT word onto stack
$the	dog jumps$	REDUCE using grammar rule
$Art	dog jumps$	SHIFT..
$Art dog	jumps$	REDUCE..
$Art Noun	jumps$	REDUCE
$NounPhrase	jumps$	SHIFT
$NounPhrase jumps	$	REDUCE
$NounPhrase Verb	$	REDUCE
$NounPhrase VerbPhrase	$	REDUCE
$Sentence	$	SUCCESS

Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.



- An ambiguous grammar can never be a LR grammar.

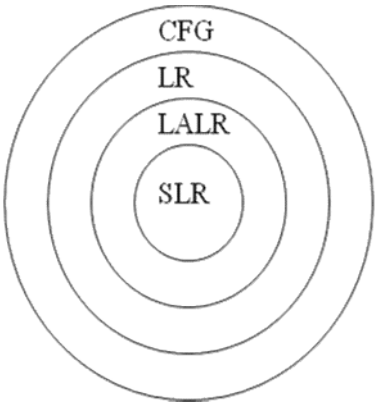
Shift-Reduce Parsers

- There are two main categories of shift-reduce parsers

1. **Operator-Precedence Parser**
   - Simple, but only a small class of grammars.
   - **LR-Parsers**

- Covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser
  - LALR – intermediate LR parser (lookhead LR parser)

SLR, LR and LALR work same, only their parsing tables are different



LR Parsers

- The most powerful shift-reduce parsing (yet efficient) is:

LR(k) parsing.

left to right scanning — right-most derivation — k lookhead (k is omitted ➔ it is 1)

- LR parsing is attractive because:
  - LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
  - The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

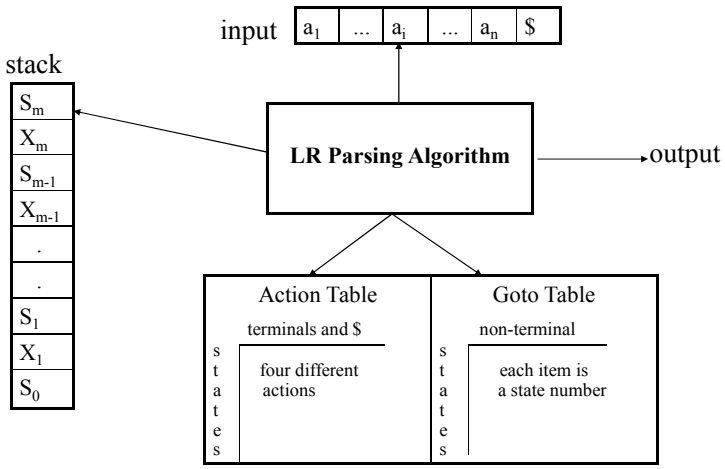
    LL(1)-Grammars $\subset$ LR(1)-Grammars
  - An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.

LR Parsers

- **LR-Parsers**
  - covers wide range of grammars.
  - SLR – simple LR parser
  - LR – most general LR parser(canonical LR)
  - LALR – intermediate LR parser (look-head LR parser)

– SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

LR Parsing Algorithm



A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$(S_o X_1 S_1 \ldots X_m S_m,\ a_i a_{i+1} \ldots a_n \$)$$

Stack Rest of Input

- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

$$X_1 \ldots X_m a_i a_{i+1} \ldots a_n \$$$

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack
   $(S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$) \rightarrow (S_o X_1 S_1 \ldots X_m S_m a_i s, a_{i+1} \ldots a_n \$)$

2. **reduce A→β** (or **rn** where n is a production number)
   – pop 2|β| (=r) items from the stack;
   – then push **A** and **s** where **s=goto[s_{m-r},A]**
   $(S_o X_1 S_1 \ldots X_m S_m, a_i a_{i+1} \ldots a_n \$) \rightarrow (S_o X_1 S_1 \ldots X_{m-r} S_{m-r} A s, a_i \ldots a_n \$)$
   – Output is the reducing production reduce A→β

3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

Reduce Action

- pop 2|β| (=r) items from the stack; let us assume that $\beta = Y_1 Y_2 \ldots Y_r$
- then push **A** and **s** where **s=goto[s_{m-r},A]**
  $(S_o X_1 S_1 \ldots X_{m-r} S_{m-r} Y_1 S_{m-r} \ldots Y_r S_m, a_i a_{i+1} \ldots a_n \$)$
  $\rightarrow (S_o X_1 S_1 \ldots X_{m-r} S_{m-r} A s, a_i \ldots a_n \$)$
- In fact, $Y_1 Y_2 \ldots Y_r$ is a handle.
  $X_1 \ldots X_{m-r} A a_i \ldots a_n \$ \Rightarrow X_1 \ldots X_m Y_1 \ldots Y_r a_i a_{i+1} \ldots a_n \$$

Constructing SLR Parsing Tables – LR(0) Item

- An LR(0) item of a grammar G is a production of G a dot at the some position of the right side.
- Ex: A → aBb Possible LR(0) Items: (four different possibility)
  A → **.**aBb
  A → a**.**Bb
  A → aB**.**b
  A → aBb**.**
- Sets of LR(0) items will be the states of action and goto table of the SLR parser.
- A collection of sets of LR(0) items (the canonical LR(0) collection) is the basis for constructing SLR parsers.
- ***Augmented Grammar*:**
  G' is G with a new production rule S'→S where S' is the new starting symbol.

The Closure Operation

- If *I* is a set of LR(0) items for a grammar G, then *closure(I)* is the set of LR(0) items constructed from I by the two rules:
  1. Initially, every LR(0) item in I is added to closure(I).

2. If $A \rightarrow \alpha.B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma$ will be in the closure(I). We will apply this rule until no more new LR(0) items can be added to closure(I).

The Closure Operation -- Example

$E' \rightarrow E$	closure({$E' \rightarrow .E$}) =
$E \rightarrow E+T$	{ $E' \rightarrow .E$ ← kernel items
$E \rightarrow T$	$E \rightarrow .E+T$
$T \rightarrow T*F$	$E \rightarrow .T$
$T \rightarrow F$	$T \rightarrow .T*F$
$F \rightarrow (E)$	$T \rightarrow .F$
$F \rightarrow id$	$F \rightarrow .(E)$
	$F \rightarrow .id$ }

Goto Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta$ in I then every item in closure({$A \rightarrow \alpha X.\beta$}) will be in goto(I,X).

Example:

I ={ $E' \rightarrow .E$, $E \rightarrow .E+T$, $E \rightarrow .T$,
$T \rightarrow .T*F$, $T \rightarrow .F$,
$F \rightarrow .(E)$, $F \rightarrow .id$ }

goto(I,E) = { $E' \rightarrow E.$, $E \rightarrow E.+T$ }

goto(I,T) = { $E \rightarrow T.$, $T \rightarrow T.*F$ }

goto(I,F) = {$T \rightarrow F.$ }

goto(I,() = { $F \rightarrow (.E)$, $E \rightarrow .E+T$, $E \rightarrow .T$, $T \rightarrow .T*F$, $T \rightarrow .F$,
$F \rightarrow .(E)$, $F \rightarrow .id$ }

goto(I,id) = { $F \rightarrow id.$ }

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.
- ***Algorithm*:**

C is { closure({$S' \rightarrow .S$}) }
repeat the followings until no more set of LR(0) items can be added to *C*.
for each I in *C* and each grammar symbol X
if goto(I,X) is not empty and not in *C*
add goto(I,X) to *C*

- goto function is a DFA on the sets in C.

The Canonical LR(0) Collection – Example

I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: E' → E.
E → E.+T

I_2: E → T.
T → T.*F

I_3: T → F.

I_4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_5: F → id.

I_6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7: T → T*.F
F → .(E)
F → .id

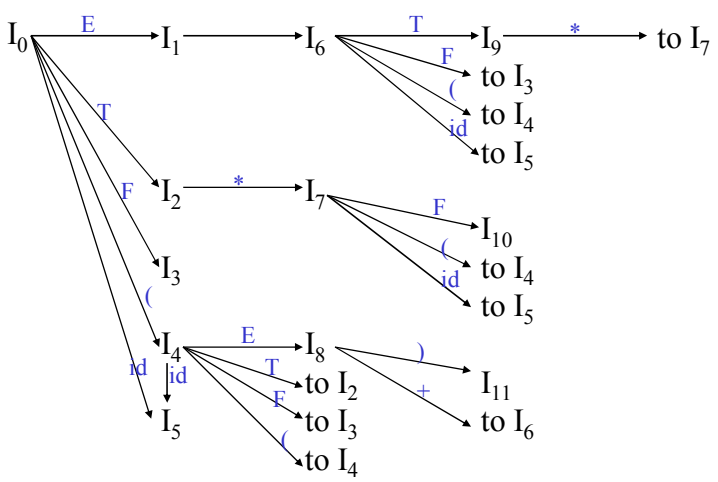
I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

I_{11}: F → (E).

Transition Diagram (DFA) of Goto Function



Constructing SLR Parsing Table

(of an augumented grammar G')

1. Construct the canonical collection of sets of LR(0) items for G'.
   $C \leftarrow \{I_0,...,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha.$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ for all a in FOLLOW(A) where $A \neq S'$.
   - If $S' \rightarrow S.$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).
3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow .S$

(SLR) Parsing Tables for Expression Grammar

1) $E \rightarrow E+T$
2) $E \rightarrow T$
3) $T \rightarrow T*F$
4) $T \rightarrow F$
5) $F \rightarrow (E)$
6) $F \rightarrow id$

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

Actions of A (S)LR-Parser – Example

stack	input	action	goto	parsing
$0	id*id+id$	[0,id]=s5		shift 5
$0id5	*id+id$	[5,*]=r6	[0,F]=3	reduce by F→id (pop 2|id| no. of symbols from stack and push F to the stack)
$0F3	*id+id$	[3,*]=r4	[0,T]=2	reduce by T→F (pop 2|F| no. of symbols from stack and push T onto the stack)
$0T2	*id+id$	[2,*]=s7		shift 7
$0T2*7	id+id$	[7,id]=s5		shift 5
$0T2*7id5	+id$	[5,+]=r6	[7,F]=10	reduce by F→id(pop 2|id| no. of symbols from stack and push F onto the stack)
$0T2*7F10	+id$	[10,+]=r3	[0,T]=2	reduce by T→T*F(pop 2 |T*F| no. of symbols from stack and push F on the stack)
$0T2	+id$	[2,+]=r2	[0,E]=1	reduce by E→T (pop 2|T| no. of symbols from stack and push E onto the stack)
$0E1	+id$	[1,+]=s6		shift 6
$0E1+6	id$	[6,id]=s5		shift 5
$0E1+6id5	$	[5,$]=r6	[6,F]=3	reduce by F→id (pop 2|id| no. of symbols from stack and push F onto the stack)
$0E1+6F3	$	[3,$]=r4	[6,F]=3	reduce by T→F (pop 2|F| no. of symbols from stack and push T onto the stack)
$0E1+6T9	$	[9,$]=r1	[0,E]=1	reduce by E→E+T (pop 2 |E+T| no. of symbols from stack and push F on the stack)
$0E1	$	accept		

Parsing Tables of Expression Grammar

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

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

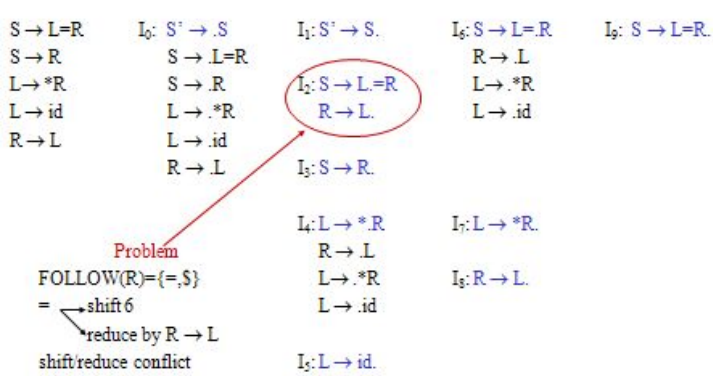
shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.
- If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal, we say that there is a **reduce/reduce conflict**.
- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Problems on SLR

1. S → SS+ | SS* | a with the string "aa+a*"
2. S → (L) | a, L → L,S | S
3. S →aSb | ab
4. S →aSbS | bSaS | ∈
5. S → E#
   E → E-T
   E → T
   T → F↑ T
   T → F
   F → (E)
   F → i
6. S → +SS | *SS | a with the string "+*aaa"
7. Show that following grammar is SLR(1) but not LL(1)
   S →SA | A
   A → a
8. X →Xb |a parse the string " abb"
9. Given the grammar A → (A) |a string "((a))"

Conflict Example



Construct parsing table for this. In this table there are 2 actions in one entry of the table which is why It is not a SLR(1) grammar.

Another example for not SLR(1) grammar:

Conflict Example2

$S \rightarrow AaAb$
$S \rightarrow BbBa$
$A \rightarrow \varepsilon$
$B \rightarrow \varepsilon$

I_0: $S' \rightarrow .S$
$S \rightarrow .AaAb$
$S \rightarrow .BbBa$
$A \rightarrow .$
$B \rightarrow .$

Problem

FOLLOW(A)={a,b}
FOLLOW(B)={a,b}

a → reduce by $A \rightarrow \varepsilon$
a → reduce by $B \rightarrow \varepsilon$
reduce/reduce conflict

b → reduce by $A \rightarrow \varepsilon$
b → reduce by $B \rightarrow \varepsilon$
reduce/reduce conflict

Problems : show that following grammars are not SLR(1) by constructing parsing table.

1. Show that $S \rightarrow S(S)S \mid \in$ not SLR(1)
2. Show that $S \rightarrow AaAb \mid BbBa$
   $A \rightarrow \in$
   $B \rightarrow \in$ is not SLR(1) but is LL(1)

UNIT IV: SYNTAX ANALYSIS – 3

SYLLABUS:

- Introduction to LR Parsing:
- Simple LR;
- More powerful LR parsers (excluding Efficient construction and compaction of parsing tables) ;
- Using ambiguous grammars;
- Parser Generators.

Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is a:
  - if the $A \rightarrow \alpha.$ in the I_i and a is FOLLOW(A)
  - In some situations, βA cannot be followed by the terminal a in a right-sentential form when $\beta\alpha$ and the state i are on the top stack. This means that making reduction in this case is not correct.

$S \rightarrow AaAb$	$S \Rightarrow AaAb \Rightarrow Aab \Rightarrow ab$	$S \Rightarrow BbBa \Rightarrow Bba \Rightarrow ba$
$S \rightarrow BbBa$		
$A \rightarrow \varepsilon$	$Aab \Rightarrow \varepsilon\, ab$	$Bba \Rightarrow \varepsilon\, ba$
$B \rightarrow \varepsilon$	$AaAb \Rightarrow Aa\, \varepsilon\, b$	$BbBa \Rightarrow Bb\, \varepsilon\, a$

LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.
- Extra information is put into a state by including a terminal symbol as a second component in an item.
- A LR(1) item is:

  $A \rightarrow \alpha.\beta,a$ where **a** is the look-head of the LR(1) item (**a** is a terminal or end-marker.)

- When β (in the LR(1) item $A \rightarrow \alpha.\beta,a$) is not empty, the look-head does not have any affect.
- When β is empty ($A \rightarrow \alpha.,a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is **a** (not for any terminal in FOLLOW(A)).
- A state will contain $A \rightarrow \alpha.,a_1$ where $\{a_1,...,a_n\} \subseteq$ FOLLOW(A)

  ...

  $A \rightarrow \alpha.,a_n$

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

closure(I) is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)

- if $A \rightarrow \alpha.B\beta,a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma,b$ will be in the closure(I) for each terminal b in FIRST(βa) .

goto operation

- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta,a$ in I then every item in **closure({A → αX.β,a})** will be in goto(I,X).

Construction of The Canonical LR(1) Collection

- ***Algorithm***:

C is { closure({S'→.S,$}) }
repeat the followings until no more set of LR(1) items can be added to ***C***.
for each I in ***C*** and each grammar symbol X
if goto(I,X) is not empty and not in ***C***
add goto(I,X) to ***C***
goto function is a DFA on the sets in C.

A Short Notation for The Sets of LR(1) Items

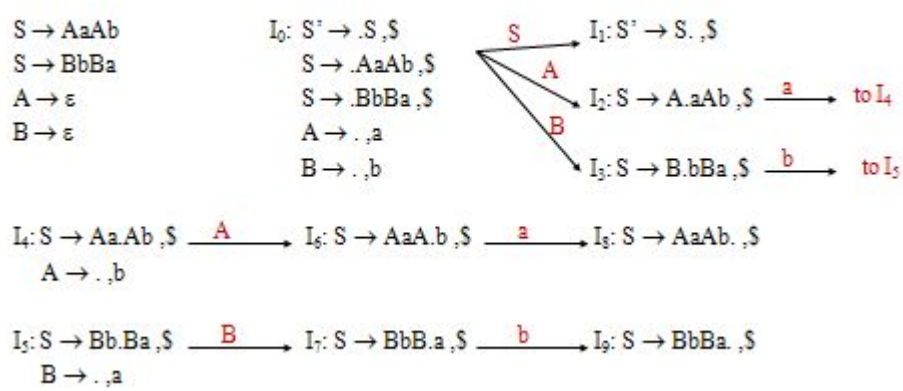
- A set of LR(1) items containing the following items

$$A \rightarrow \alpha.\beta,a_1$$
$$\ldots$$
$$A \rightarrow \alpha.\beta,a_n$$

can be written as

$$A \rightarrow \alpha.\beta,a_1/a_2/\ldots/a_n$$

Canonical LR(1) Collection -- Example



Canonical LR(0) Collection

- S' → S
S → L=R

S → R
L→ *R
L → id
R → L

I_0: S' → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

I_1: S' → S.

I_2: S → L.=R
R → L.

I_3: S → R.

I_4: L → *.R
R → .L
L→ .*R
L → .id

I_5: L → id.

I_6: S → L=.R
R → .L
L → .*R
L → .id

I_7: L → *R.

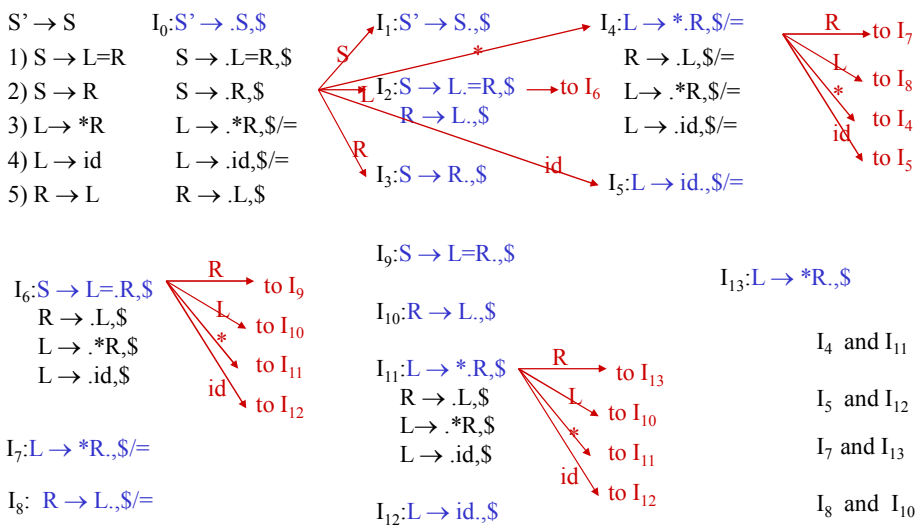
I_8: R → L.

I_9: S → L=R.

SLR(1) Parsing table

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6/r5	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s5	s4				10	9
7			r3	r3			
8			r5	r5			
9				r1			

Canonical LR(1) Collection – Example2



Construction of LR(1) Parsing Tables

1. Construct the canonical collection of sets of LR(1) items for G'. $C \leftarrow \{I_0,...,I_n\}$
2. Create the parsing action table as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta,b$ in I_i and $goto(I_i,a)=I_j$ then action[i,a] is ***shift j.***
   - If $A \rightarrow \alpha.,a$ is in I_i, then action[i,a] is ***reduce*** $A \rightarrow \alpha$ where $A \neq S'$.
   - If $S' \rightarrow S.,\$$ is in I_i, then action[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not LR(1).
3. Create the parsing goto table
   - for all non-terminals A, if $goto(I_i,A)=I_j$ then goto[i,A]=j
4. All entries not defined by (2) and (3) are errors.

Initial state of the parser contains $S' \rightarrow .S,\$$

LR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			
10				r5			
11	s12	s11				10	13
12				r4			
13				r3			

no shift/reduce or
no reduce/reduce conflict

⇓

so, it is a LR(1) grammar

LALR Parsing Tables

- **LALR** stands for **LookAhead LR.**
- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar G are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- ***yacc*** creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

Creating LALR Parsing Tables

Canonical LR(1) Parser ➔ LALR Parser
shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrik process does not produce a **shift/reduce** conflict.

The Core of A Set of LR(1) Items

- The core of a set of LR(1) items is the set of its first component.

Ex: $S \rightarrow L.=R,\$$ ➔ $S \rightarrow L.=R$ ⟵ Core
$R \rightarrow L.,\$$ $R \rightarrow L.$

- We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

I_1: $L \rightarrow id.,=$ ➔ A new state: I_{12}: $L \rightarrow id.,=$
$L \rightarrow id.,\$$

I_2:L → id.,$ have same core, merge them

- We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.
- In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

- Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.
- Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

  $C=\{I_0,...,I_n\} \rightarrow C'=\{J_1,...,J_m\}$ where $m \leq n$
- Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
  - Note that: If $J=I_1 \cup ... \cup I_k$ since $I_1,...,I_k$ have same cores

    → cores of goto(I_1,X),...,goto(I_2,X) must be same.
  - So, goto(J,X)=K where K is the union of all sets of items having same cores as goto(I_1,X).
- If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

Shift/Reduce Conflict

- We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.
- Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

  $A \rightarrow \alpha.,a$ and $B \rightarrow \beta.a\gamma,b$
- This means that a state of the canonical LR(1) parser must have:

  $A \rightarrow \alpha.,a$ and $B \rightarrow \beta.a\gamma,c$

But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)

Reduce/Reduce Conflict

- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

I_1 : $A \rightarrow \alpha.,a$ $\quad$ I_2: $A \rightarrow \alpha.,b$

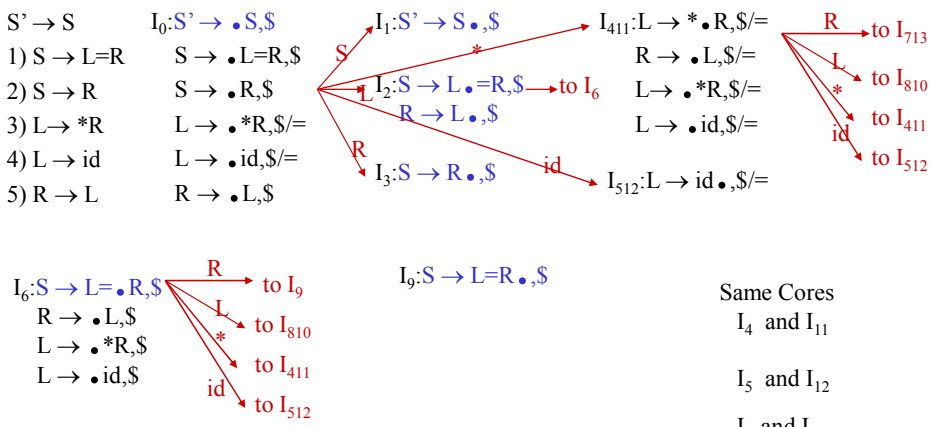
$B \rightarrow \beta.,b$ $\quad$ $B \rightarrow \beta.,c$

$\Downarrow$

I_{12}: $A \rightarrow \alpha.,a/b$ $\quad$ ➔ reduce/reduce conflict

$B \rightarrow \beta.,b/c$

Canonical LALR(1) Collection – Example2



LALR(1) Parsing Tables – (for Example2)

	id	*	=	$	S	L	R
0	s5	s4			1	2	3
1				acc			
2			s6	r5			
3				r2			
4	s5	s4				8	7
5			r4	r4			
6	s12	s11				10	9
7			r3	r3			
8			r5	r5			
9				r1			

no shift/reduce or
no reduce/reduce conflict

$\Downarrow$

so, it is a LALR(1) grammar

Using Ambiguous Grammars

- All grammars used in the construction of LR-parsing tables must be un-ambiguous.
- Can we create LR-parsing tables for ambiguous grammars ?
  - Yes, but they will have conflicts.
  - We can resolve these conflicts in favor of one of them to disambiguate the grammar.

- At the end, we will have again an unambiguous grammar.

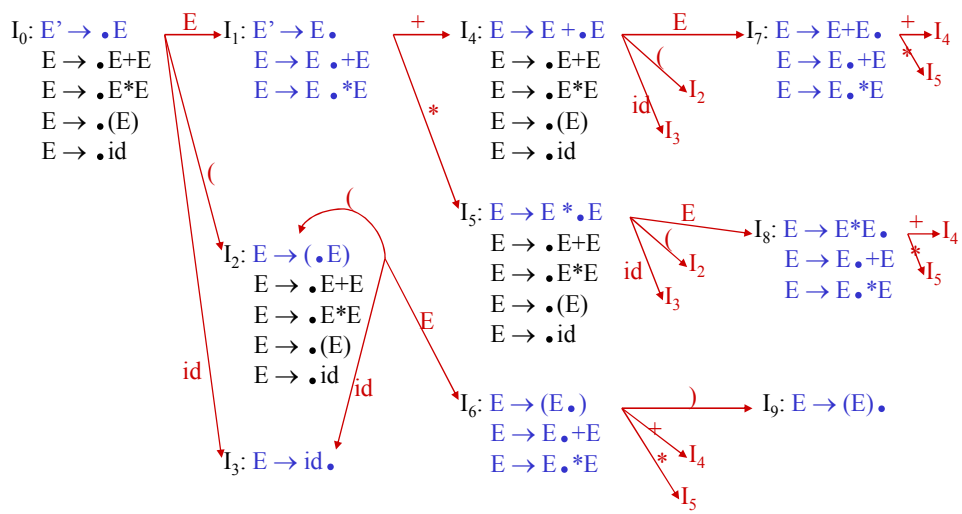
- Why we want to use an ambiguous grammar?
  - Some of the ambiguous grammars are **much natural**, and a corresponding unambiguous grammar can be very complex.
  - Usage of an ambiguous grammar may **eliminate unnecessary reductions**.

- Ex.

$E \rightarrow E+E \mid E*E \mid (E) \mid id$ ➔

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid id$$

Sets of LR(0) Items for Ambiguous Grammar



SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $, +, *,) }

State I_7 has shift/reduce conflicts for symbols + and *.

$I_0 \xrightarrow{E} I_1 \xrightarrow{+} I_4 \xrightarrow{E} I_7$

when current token is +

shift ➔ + is right-associative

reduce ➔ + is left-associative

when current token is *

shift ➔ * has higher precedence than +

reduce ➔ + has higher precedence than *

SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $, +, *,) }

State I_8 has shift/reduce conflicts for symbols + and *.

$I_0 \xrightarrow{E} I_1 \xrightarrow{*} I_5 \xrightarrow{E} I_7$

when current token is *

shift ➔ * is right-associative

reduce ➔ * is left-associative

when current token is +

shift ➔ + has higher precedence than *

reduce ➔ * has higher precedence than +

SLR-Parsing Tables for Ambiguous Grammar

	Action						Goto
	id	**+**	*****	**(**	**)**	**$**	**E**
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.
- Errors are never detected by consulting the goto table.
- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.
- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.
- The SLR and LALR parsers may make several reductions before announcing an error.
- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state **s** with a goto on a particular nonterminal **A** is found. (Get rid of everything from the stack before this state s).
- Discard zero or more input symbols until a symbol **a** is found that can legitimately follow A.
  - The symbol a is simply in FOLLOW(A), but this may not work for all situations.
- The parser stacks the nonterminal **A** and the state **goto[s,A]**, and it resumes the normal parsing.
- This nonterminal A is normally is a basic programming block (there can be more than one choice for A).
  - stmt, expr, block, ...

Phrase-Level Error Recovery in LR Parsing

- Each empty entry in the action table is marked with a specific error routine.
- An error routine reflects the error that the user most likely will make in that case.
- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
  - missing operand
  - unbalanced right parenthesis

PART-B

UNIT V: SYNTAX-DIRECTED DEFINITIONS

SYLLABUS:

- Syntax-directed definitions;
- Evaluation orders for SDDs;
- Applications of syntax-directed translation;
- Syntax-directed translation schemes

Overview

input → parse tree → dependency graph → attribute evaluation order

- ✓ Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- ✓ Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- ✓ Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - in fact, they may perform almost any activities.
- ✓ An attribute may hold almost anything.
  - a string, a number, a memory location, a complex record.

  Attributes for expressions:

  type of value: int, float, double, char, string,…

  type of construct: variable, constant, operations, …

  Attributes for constants: values

  Attributes for variables: name, scope
  - Attributes for operations: arity, operands, operator,…
- ✓ When we associate semantic rules with productions, we use two notations:
  - **Syntax-Directed Definitions**
  - **Translation Schemes**
- ✓ **Syntax-Directed Definitions:**
  - give high-level specifications for translations
  - Hide many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
- ✓ **Translation Schemes:**
  - Indicate the order of evaluation of semantic actions associated with a production rule.
  - In other words, translation schemes give a little bit information about implementation details.

Syntax directed definition (SDD) :

- ✓ To translate a programming language construct compiler has to keep track of many quantities such as the type of the construct, location of the first instruction in target code or the number of instructions generated.

- ✓ A formalist called as syntax directed definition is used fort specifying translations for programming language constructs.
- ✓ A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semantic rules

Definition of (syntax Directed definition) SDD :

SDD is a generalization of CFG in which each grammar productions $X \rightarrow \alpha$ is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)

Where a is an attributes obtained from the function f.

- A syntax-directed definition is a generalization of a context-free grammar in which:
  - Each grammar symbol is associated with a set of attributes.
  - This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited** attributes of that grammar symbol.
  - Each production rule is associated with a set of semantic rules.
- *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.
- This *dependency graph* determines the evaluation order of these semantic rules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

The two attributes for non terminal are :

1) **<u>synthesized attribute (S-attribute)</u> : (↑)**

An attribute is said to be synthesized attribute if its value at a parse tree node is determined from attribute values at the <u>children of the node</u>

2) **<u>Inherited attribute :</u> (→,↑)**

An inherited attribute is one whose value at parse tree node is determined in terms of attributes at the <u>parent and | or siblings of that node.</u>

- ❖ The attribute can be string, a number, a type, a, memory location or anything else.
- ❖ The parse tree showing the value of attributes at each node is called an <u>annotated parse tree</u>.

The process of computing the attribute values at the node is called annotating or decorating the parse tree.

Terminals can have synthesized attributes, but not inherited attributes.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

- Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Ex1:

1) <u>Synthesized Attributes :</u>

Ex: Consider the CFG :

S→ EN
E→ E+T
E→E-T
E→ T
T→ T*F
T→T/F
T→F
F→(E)
F→digit
N→;

<u>Solution :</u>

The syntax directed definition can be written for the above grammar by using semantic actions for each production.

Production rule	Semantic actions
S →EN	S.val=E.val
E →E1+T	E.val =E1.val + T.val
E→E1-T	E.val = E1.val – T.val
E→T	E.val =T.val
T→T*F	T.val = T.val * F.val
T→T|F	T.val =T.val | F.val
F→(E)	F.val =E.val
T →F	T.val =F.val
F→digit	F.val =digit.lexval
N→;	can be ignored by lexical Analyzer as; is terminating Symbol.

For the Non-terminals E,T and F the values can be obtained using the attribute "Val".
The taken digit has synthesized attribute "lexval".
In S→EN, symbol S is the start symbol. This rule is to print the final answer of expressed.
Following steps are followed to Compute S attributed definition.

1. Write the SDD using the appropriate semantic actions for corresponding production rule of the given Grammar.
2. The annotated parse tree is generated and attribute values are computed. The Computation is done in bottom up manner.
3. The value obtained at the node is supposed to be final output.

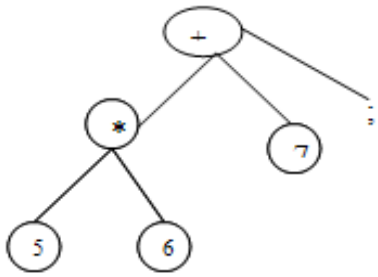
PROBLEM 1:

Consider the string 5*6+7; Construct Syntax tree, parse tree and annotated tree.

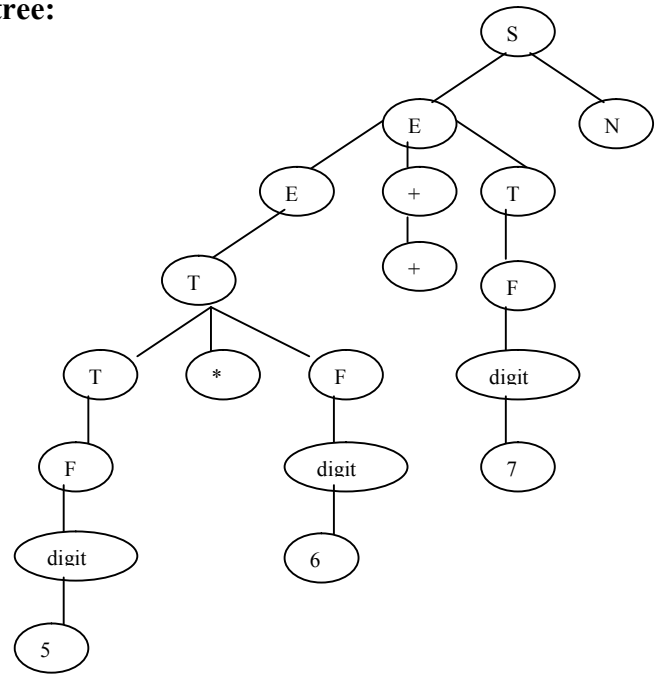
Solution :

The corresponding annoted parse tree is shown below for the string 5*6+7;

Syntax tree:



Parse tree:



The corresponding annoted parse tree is shown below for the string 5*6+7;

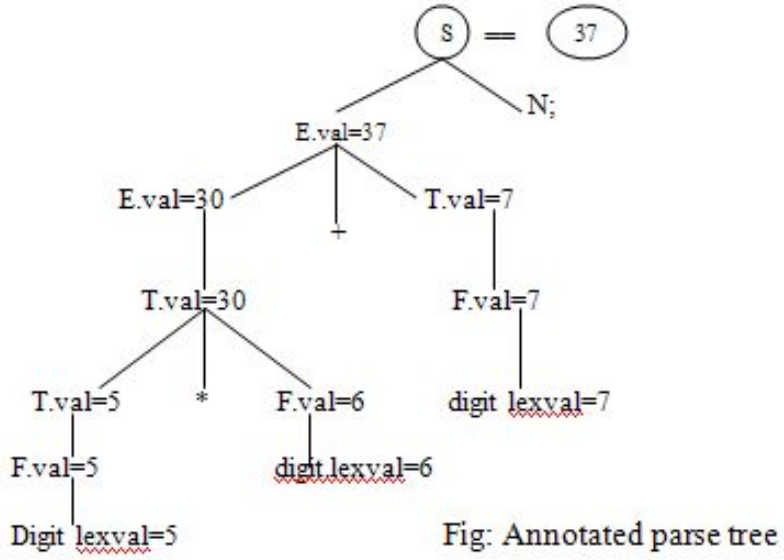


Fig: Annotated parse tree

Advantages: SDDs are more readable and hence useful for specifications

Disadvantages: not very efficient.

Ex2:

PROBLEM : Consider the grammar that is used for Simple desk calculator. Obtain the Semantic action and also the annotated parse tree for the string 3*5+4n.

L→En
E→E1+T
E→T
T→T1*F
T→F
F→(E)
F→digit

Solution :

Production rule	Semantic actions
L→En	L.val=E.val
E→E1+T	E.val=E1.val + T.val
E→T	E.val=T.val
T→T1*F	T.val=T1.val*F.val
T→F	T.val=F.val
F→(E)	F.val=E.val
F→digit	F.val=digit.lexval

The corresponding annotated parse tree U shown below, for the string 3*5+4n.

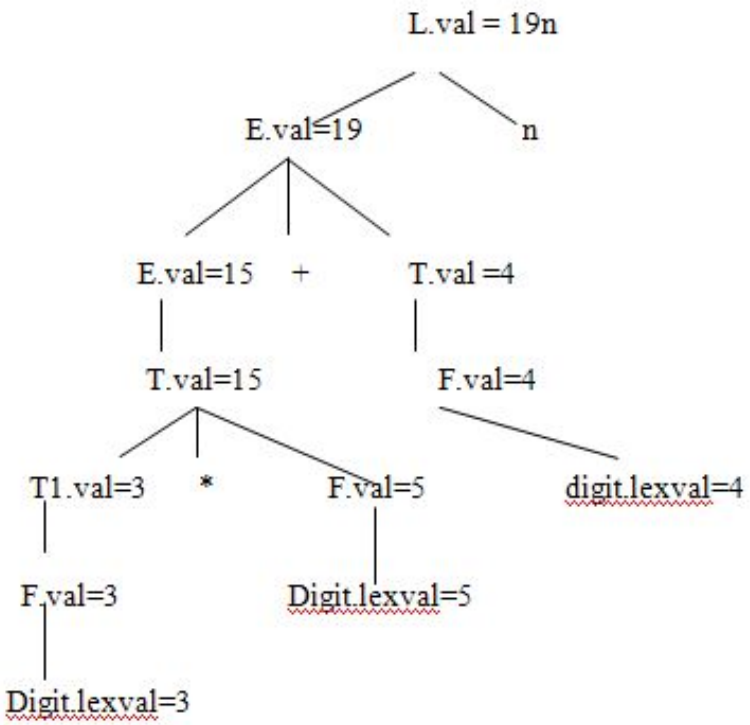


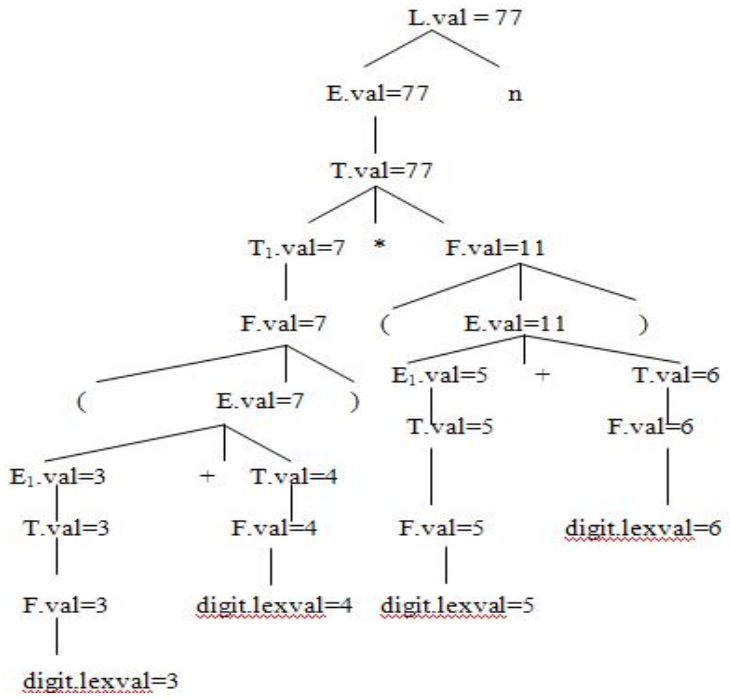
Fig:Annotated parse tree

Exercise :

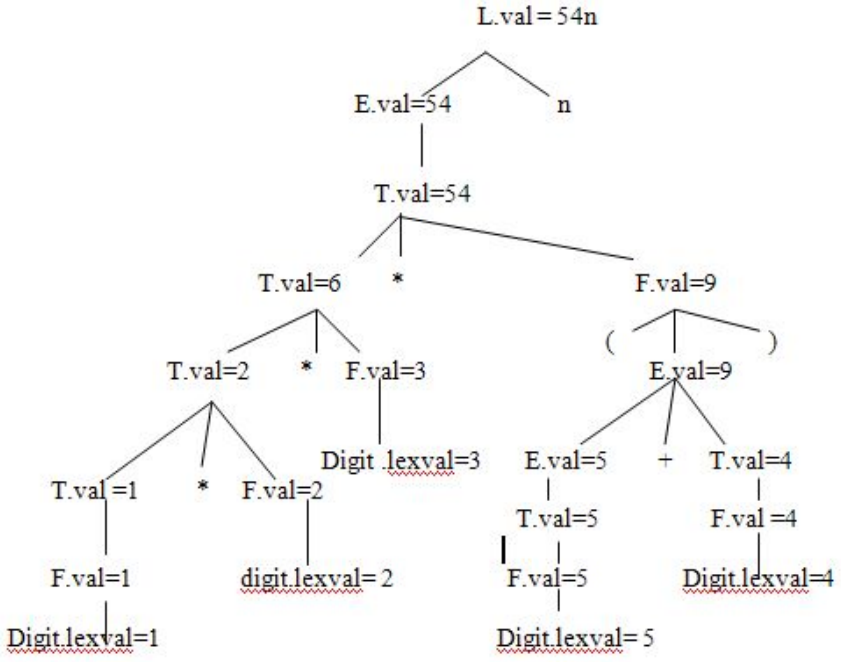
For the SDD of the problem 1 give annotated parse tree for the Following expressions

a) (3+4)*(5+6)n
b) 1*2*3*(4+5)n
c) (9+8*(7+6)+5)*4n

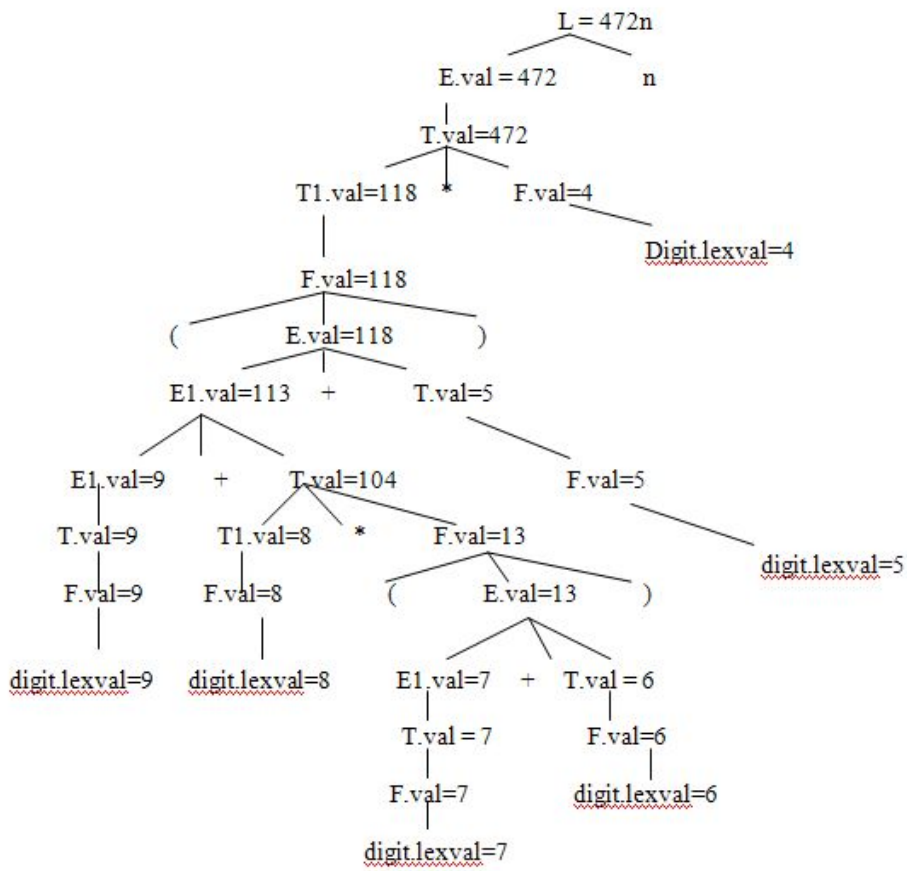
Solution: a)


L.val = 77
E.val=77
n
T.val=77
T1.val=7
*
F.val=11
F.val=7
(
E.val=11
)
(
E.val=7
)
E1.val=5
+
T.val=6
T.val=5
F.val=6
E1.val=3
+
T.val=4
T.val=3
F.val=4
F.val=5
digit.lexval=6
F.val=3
digit.lexval=4
digit.lexval=5
digit.lexval=3

b)

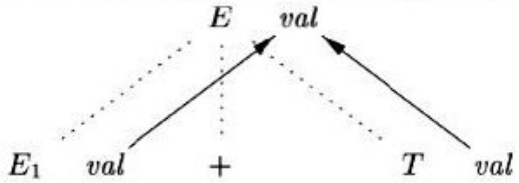

L.val = 54n
E.val=54
n
T.val=54
T.val=6
*
F.val=9
(
)
T.val=2
*
F.val=3
E.val=9
Digit .lexval=3
E.val=5
+
T.val=4
T.val =1
*
F.val=2
T.val=5
F.val =4
F.val=1
digit.lexval= 2
F.val=5
Digit.lexval=4
Digit.lexval=1
Digit.lexval= 5

c)



Dependency Graphs

Figure 5.6. *E.val* is synthesized from E_1.*val* and E_2.*val*



Note: $E_2.val$ should be $T.val$ in Figure 5.6

Dependency graph and topological sort:

1. For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.
2. If a semantic rule associated with a production p defines the value of synthesized attribute A.b in terms of the value of X.c. Then the dependency graph has an edge from X.c to A.b .
3. If a semantic rule associated with a production p defines the value of inherited attribute B.c in terms of the value X.a. Then , the dependency graph has an edge from X.a to B.c.

Evaluation Orders for SDD's

- Dependency graphs – are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.
- A dependency graph depicts the flow of information among the attribute instances in a particular parse tree.
  - An edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.

Edges express constraints implied by the semantic rules

- A *dependency graph* is a directed graph depicting the relationships among inherited and synthesized attributes in a parse tree.

Constructing a dependency graph:

```
for each node N in a parse tree
    for each attribute A of node N
        construct a node labelled A in the dependency graph

for each node N in a parse tree
    for each semantic function A=f(A1,A2,...,Ak) at node N
        for i := 1 to k
            construct an edge from node Ai to node A in the graph
```

2) Inherited attributes :

Consider an example and compute the inherited attributes, annotate the parse tree for the computation of inherited attributes for the given string int a, b, c

Ex:

S→TL

T→int

T→float

T→char

T→double

L→L,id

L →id

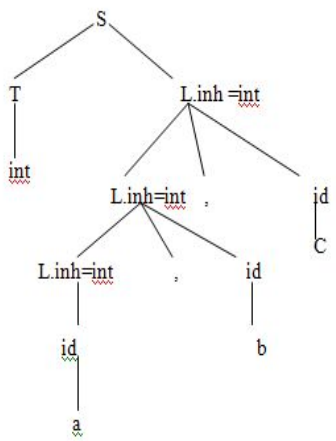
The steps are to be followed are:

1) Construct the syntax directed definition using semantic action.
2) Annotate the parser tree with inherited attributes by processing in top down fashion.

The SDD is given below:

Production rule	Semantic actions
S → TL	L.inh =T.type
T→int	T.type = int
T→float	T.type =float
T→char	T.type =char
T→double	T.type=double
L→L,id	T.type = L, inh Add–type (id.entry,L.inh)
L→id	Add – type (id.entry,L.inh)

string int a, b,c



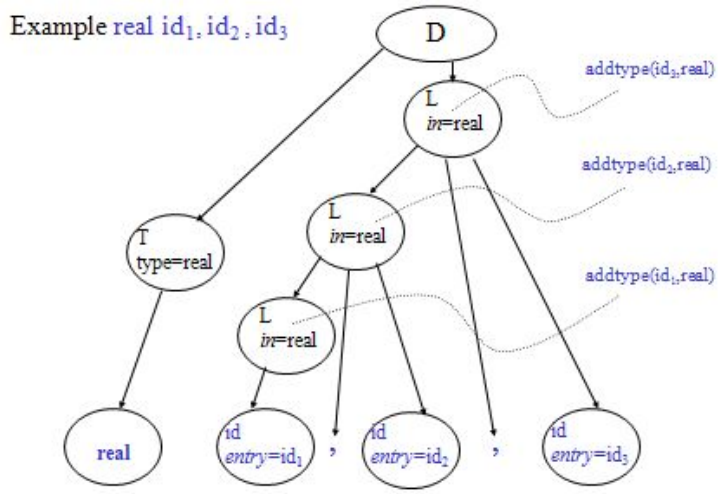


Fig: Annotated parse tree.

String float id1, id2, id3

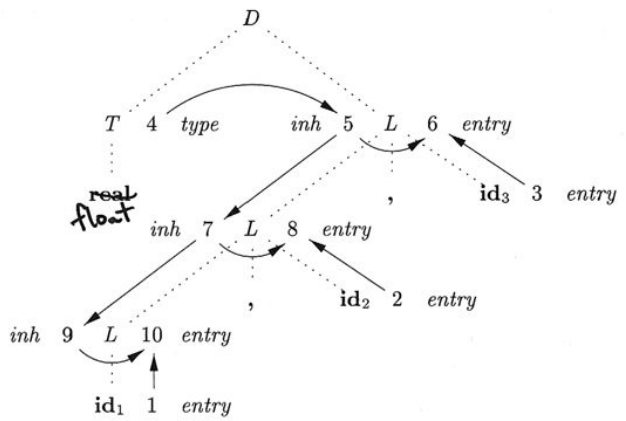


Figure 5.9: Dependency graph for a declaration **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

Ex2: PROBLEMS: consider the following context free grammar for evaluating arithmetic expressions with operator *

T→ FT'

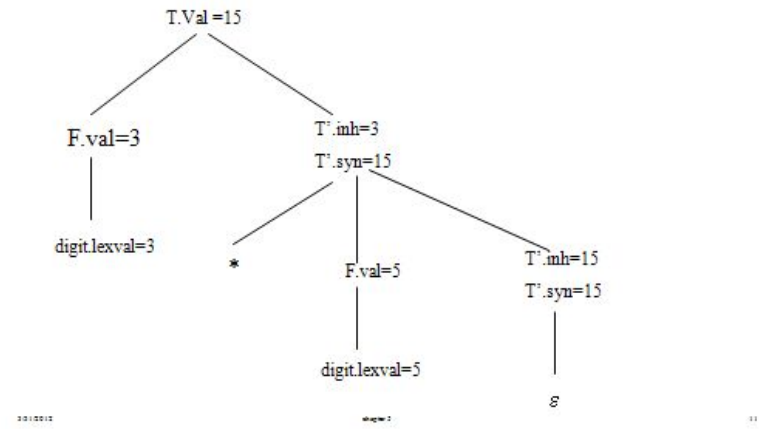
T' → *FT'

T' →∈

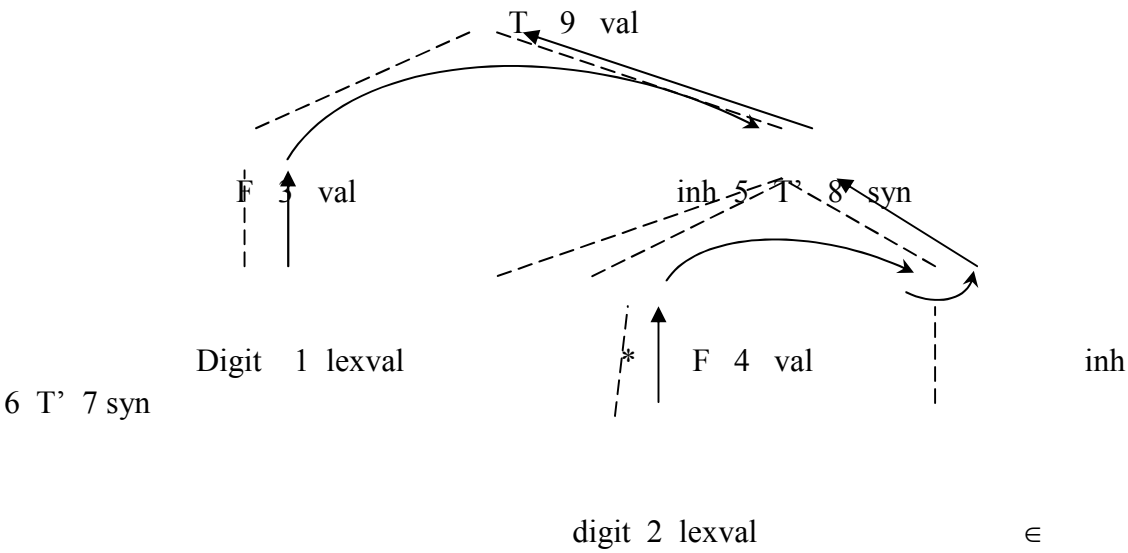
F →digit

Production	Semantic Rules
T → FT'	T'.inh = F.val T.val = T'.syn
T' → *FT'	T'.inh = T'.inh * F.val T'.syn = T'.syn
T →ε	T'.syn=T'.inh
F → **digit**	F.val = **digit**.lexval

Annotated Parse Tree for 3 * 5



Dependency graph above example:



A topological sort of the above graph gives the order of evaluation of the SDD. One of the topological sort order is (1,2,3,4,5,6,7,8 and 9) another topological sort order is (1,2,5,2,4,6,7,8,9)

Advantages: dependency graph helps in computing the order of evaluation of the attributes

Disadvantage: it cannot give the order of evaluation of attributes if there is a cycle formation in the graph. However, this disadvantage can be overcome by using S – attributed and L – attributed definitions.

Problems on SDD, Annotated parse tree, Dependency graph, evaluation of order:

1. E →TE' E' → +TE' E' → T →FT' T' → *FT' T' → F →(E) F → id String (i) "id +id*id" (ii) " (id + id* id)"	2. T → BC B → int B → float C → [num]C C → String (i) "int[2][3]" (note: int [2][3] should be passed as **array(2, array(3, integer))**) (ii) "float [3]" (iii) "float [3][3][2]"

S-Attributed Definitions

- Syntax-directed definitions are used to specify syntax-directed translations.
- To create a translator for an arbitrary syntax-directed definition can be difficult.
- We would like to evaluate the semantic rules during parsing (i.e. in a single pass, we will parse and we will also evaluate semantic rules during the parsing).
- We will look at two sub-classes of the syntax-directed definitions:
  - **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.
  - **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.
- To implement S-Attributed Definitions and L-Attributed Definitions are easy (we can evaluate semantic rules in a single pass during the parsing).
- Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

L-Attributed Definitions

- S-Attributed Definitions can be efficiently implemented.
- We are looking for a larger (larger than S-Attributed Definitions) subset of syntax-directed definitions which can be efficiently evaluated.

  ➔ L-Attributed Definitions
- L-Attributed Definitions can always be evaluated by the depth first visit of the parse tree.

This means that they can also be evaluated during the parsing

- A syntax-directed definition is **L-attributed** if each inherited attribute of X_j, where $1 \leq j \leq n$, on the right side of $A \rightarrow X_1X_2...X_n$ depends only on:
  1. The attributes of the symbols $X_1,...,X_{j-1}$ to the left of X_j in the production and
  2. the inherited attribute of A
- Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (not to synthesized attributes).

L-attributed definitions

- A syntax-directed definition is L-attributed if each inherited attribute of X_j, $1<=j<=n$, on the right side of $A::=X_1X_2...X_n$, depends only on
  - the attributes of $X_1,X_2,...,X_{j-1}$ to the left of X_j in the production
  - the inherited attributes of A

L-attributed definition

Production	Semantic rules
D::=T L	L.in:=T.type
T::= **int**	T.Type:=integer
T::=**real**	T.type:=real
L::=L1 **,id**	L1.in := L.in Addtype(id.entry,L.in)
L::=**id**	Addtype(id.entry,L.in)

Non L-attributed definition

Production	Semantic rules
A::=L M	L.i = A.i M.i = L.s A.s = M.s
A ::= Q R	R.i = A.i Q.i = R.s A.s = Q.s

Semantic Rules with Controlled Side effects

- Permit incidental side effects the do not constrain attribute evaluation
- Constrain the allowable evaluation orders, so that the same translation is produced foer any allowable order.
  - Ex: For production L→ En Semantic Rule is print(E.val)

Implementing S-attributed definitions

Implementation of a desk calculator with an LR parser (when a number is shifted onto symbol stack, its value is shifted onto val stack)

production	Code fragment
E' ::= E	Print(val[top])
E ::= E1 + T	v=val[top-2]+val[top]; top-=2; val[top]=v;
E ::= T	
T ::= T1 * F	v=val[top-2]*val[top]; top-=2; val[top]=v;
T ::= F	
F ::= (E)	v=val[top-1]; top-=2; val[top]=v
F ::= n	

Applications of Syntax-Directed Translation

- Construction of syntax Trees
  - The nodes of the syntax tree are represented by objects with a suitable number of fields.
  - Each object will have an ***op*** field that is the label of the node.
  - The objects will have additional fields as follows
    - If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function ***Leaf (op, val)*** creates a leaf object.
    - If nodes are viewed as records, the Leaf returns a pointer to a new record for a leaf.

- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function ***Node*** takes two or more arguments:

Node (op , c1,c2,…...ck) creates an object with first field ***op*** and ***k*** additional fields for the ***k*** children ***c1,c2,…...ck***

SDD- To construct syntax tree for a simple expression

Production	**Semantic Rules**
$E \rightarrow E_1 + T$	E.node = **new** Node ('+', E_1 .node, T.node)
$E \rightarrow E_1 - T$	E.node = **new** Node ('-', E_1 .node, T.node)
$E \rightarrow T$	E. node= T.node
$T \rightarrow (E)$	T.node = E.node
T → **id**	T.node= **new** Leaf (**id**, **id**.entry)
T → **num**	T.node= **new** Leaf (**num**, **num**.val)

This is an example for S-attributed definition

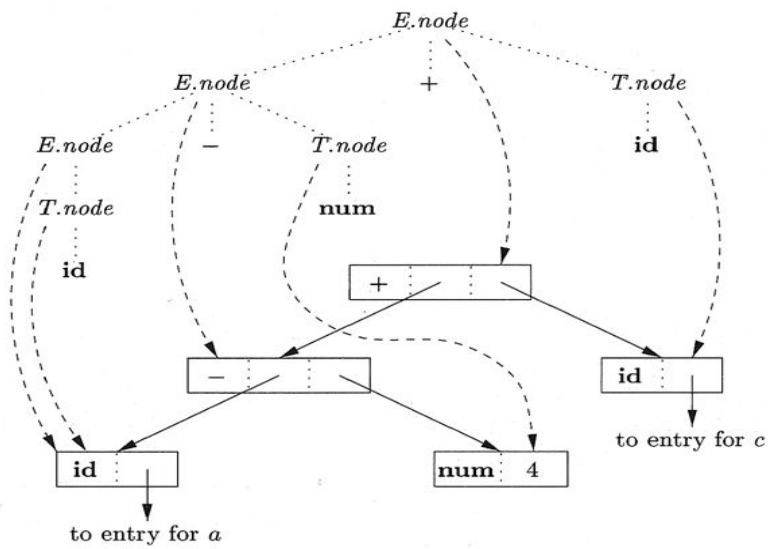


Figure 5.11: Syntax tree for $a - 4 + c$

1) $p_1 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}a);$
2) $p_2 = \textbf{new}\ Leaf(\textbf{num}, 4);$
3) $p_3 = \textbf{new}\ Node('-', p_1, p_2);$
4) $p_4 = \textbf{new}\ Leaf(\textbf{id}, entry\text{-}c);$
5) $p_5 = \textbf{new}\ Node('+', p_3, p_4);$

Figure 5.12: Steps in the construction of the syntax tree for $a - 4$

L-attributed definition for Simple Expression

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (\ E\ )$	$T.node = E.node$
6)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.13: Constructing syntax trees during top-down parsing

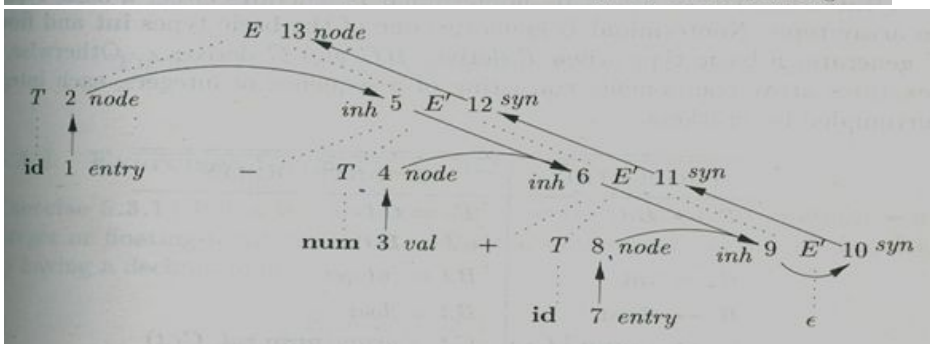


Figure 5.14: Dependency graph for $a - 4 + c$, with the SDD of Fig. 5.13

Syntax-Directed Translation Schemes

A SDT scheme is a context-free grammar with program fragments embedded within production bodies .The program fragments are called semantic actions and can appear at any position within the production body.

Any SDT can be implemented by first building a parse tree and then pre-forming the actions in a left-to-right depth first order. i.e during preorder traversal.

The use of SDT's to implement two important classes of SDD's

1. If the grammar is LR parsable, then SDD is S-attributed.
2. If the grammar is LL parsable, then SDD is L-attributed.

Postfix Translation Schemes

The postfix SDT implements the desk calculator SDD with one change: the action for the first production prints the value. As the grammar is LR, and the SDD is S-attributed.

L → E n {print(E.val);}

$E \rightarrow E_1 + T$ { $E.val = E_1.val + T.val$ }

$E \rightarrow E_1 - T$ { $E.val = E_1.val - T.val$ }

$E \rightarrow T$ { $E.val = T.val$ }

$T \rightarrow T_1 * F$ { $T.val = T_1.val * F.val$ }

$T \rightarrow F$ { $T.val = F.val$ }

$F \rightarrow (E)$ { $F.val = E.val$ }

F → digit { F.val = digit.lexval }

Syntax Trees

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.10: Constructing syntax trees for simple expressions

Postfix SDT's

- Leftmost: the leftmost nonterminal is always chosen for expansion at each step of derivation.

L-attributed SDT's

- Shows a graphical depiction of a derivation.

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Figure 2.10: Syntax-directed definition for infix to postfix translation

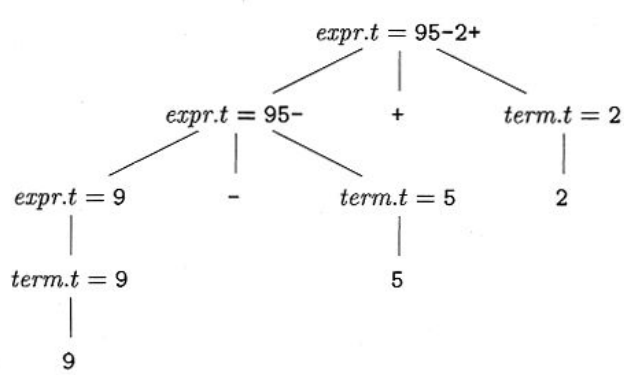


Figure 2.9: Attribute values at nodes in a parse tree

Bottom-Up Evaluation of S-Attributed Definitions

- **We put the values of the synthesized attributes of the grammar symbols into a parallel stack.**
  - **When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symbol X.**
- **We evaluate the values of the attributes during reductions.**

A → XYZ A.a=f(X.x,Y.y,Z.z) where all attributes are synthesized.

Production	**Semantic Rules**
L → E	**return** {print(stack[top-1].val); top =top – 1;}
E → E_1 + T	{stack[top-2].val = stack[top-2].val + stack[top].val; top = top – 2;}
E → T	
T → T_1 * F	{stack[top-2].val = stack[top-2].val * stack[top].val; top = top – 2;}
T → F	
F → (E)	{ stack[top-2].val = stack[top-1].val ; top = top – 2;}
F → **digit**	

- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
- At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Translation Schemes

- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated?).
- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules (when the semantic rules associated with a production should be evaluated?).
- A **translation scheme** is a context-free grammar in which:
- attributes are associated with the grammar symbols and
- semantic actions enclosed between braces {} are inserted within the right sides of productions.
- *Ex:* A → { ... } X { ... } Y { ... }

Semantic Actions

- When designing a translation scheme, some restrictions should be observed to ensure that an attribute value is available when a semantic action refers to that attribute.
- These restrictions (motivated by L-attributed definitions) ensure that a semantic action does not refer to an attribute that has not yet computed.
- In translation schemes, we use *semantic action* terminology instead of *semantic rule* terminology used in syntax-directed definitions.
- The position of the semantic action on the right side indicates when that semantic action will be evaluated.

Translation Schemes for S-attributed Definitions

- If our syntax-directed definition is S-attributed, the construction of the corresponding translation scheme will be simple.
- Each associated semantic rule in a S-attributed syntax-directed definition will be inserted as a semantic action into the end of the right side of the associated production.

<u>Production</u> <u>Semantic Rule</u>

$E \rightarrow E_1 + T$ $E.val = E_1.val + T.val$ ➔ a production of a syntax directed definition

⇓

$E \rightarrow E_1 + T$ { $E.val = E_1.val + T.val$ } ➔ the production of the corresponding translation scheme

SDT for infix –to- prefix translation during parsing

L➔ E n

E → {print('+');} $E_1 + T$

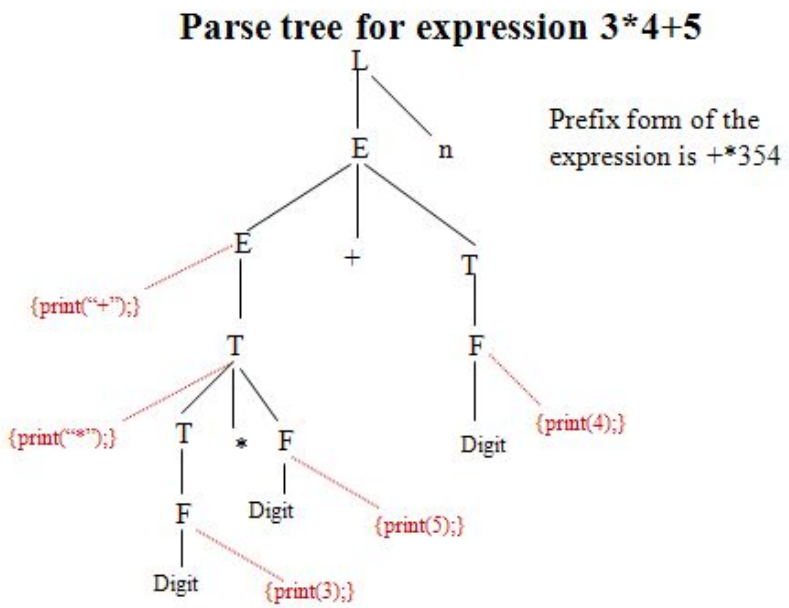
E → T

T → {print('*');} $T_1 * F$

T → F

F → (E)

F → **digit** { print(**digit**.lexval); }



A Translation Scheme Example

- **A simple translation scheme that converts infix expressions to the corresponding postfix expressions.**

  E → T R

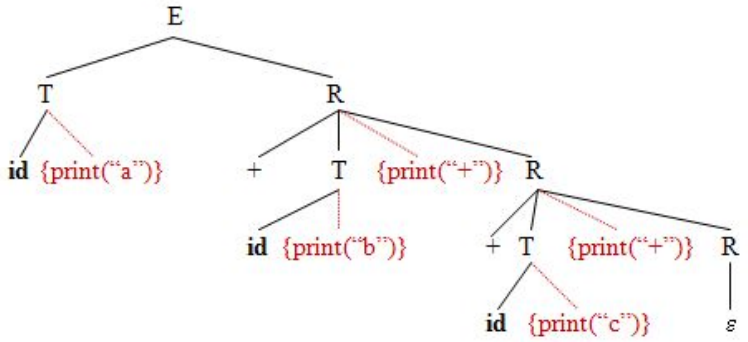
  R → + T { print("+") } R_1

  R → ε

  T → id { print(id.name) }

a+b+c ➔ ab+c+

infix expression **postfix expression**



The depth first traversal of the parse tree (executing the semantic actions in that order) will produce the postfix representation of the infix expression.

Inherited Attributes in Translation Schemes

- If a translation scheme has to contain both synthesized and inherited attributes, we have to observe the following rules:

1. An inherited attribute of a symbol on the right side of a production must be computed in a semantic action before that symbol.
2. A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.
3. A synthesized attribute for the non-terminal on the left can only be computed after all attributes it references have been computed (we normally put this semantic action at the end of the right side of the production).
4. With a L-attributed syntax-directed definition, it is always possible to construct a corresponding translation scheme which satisfies these three conditions (This may not be possible for a general syntax-directed translation).

A Translation Scheme with Inherited Attributes

D → T **id** { addtype(**id**.entry,T.type), L.in = T.type } L

T → **int** { T.type = integer }

T → **real** { T.type = real }

L → **id** { addtype(**id**.entry,L.in), L_1.in = L.in } L_1

L → ε

This is a translation scheme for an L-attributed definition.

UNIT VI: INTERMEDIATE-CODE GENERATION

SYLLABUS

- Variants of syntax trees;
- Three-address code;
- Translation of expressions;
- Control flow; Back patching;
- Switch statements;
- Procedure calls.

Background

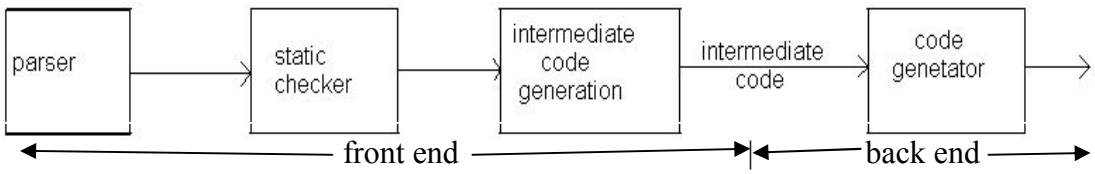


Fig 6.1: Logical structure of the compiler frond end

Static checking includes type checking, which ensures that operands are applied to compatible operands. It also includes any syntactic checks that remain after parsing.

Ex: Static checking assures that a break statement in C is enclosed within a while, for or switches statement; otherwise an error message is issued.

Intermediate representation

A complier may construct a sequence of intermediate representation as in fig.

Fig6.2: A compiler might use a sequence of intermediate representation

High level representations are close to source language and low level representations are close to the target machine.

There are three types of intermediate representation:-

1. Syntax Trees
2. Postfix notation
3. Three Address Code

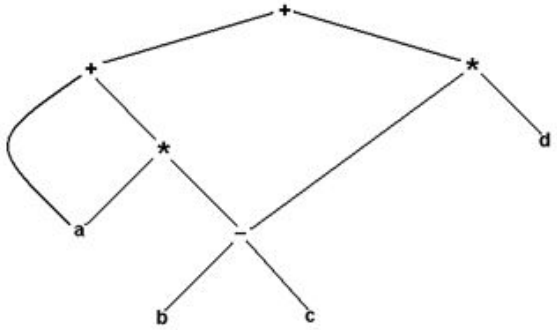
6.1 Variants of Syntax Trees

- Nodes of syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct.
- A directed acyclic graph (DAG) for an expression identifies the *common subexpressions* of the expression. (*subexpressions* that appears more than once)
- A DAG as leaves corresponding to **atomic operands** and interior codes corresponding to **operators.** A node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree.

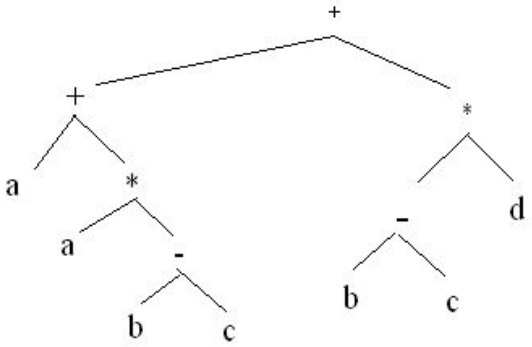
6.1.1 Directed Acyclic Graphs for Expressions

Eg1: Following Figure shows a *dag* for the expression **a + a * (b - c) + (b - c) * d**.

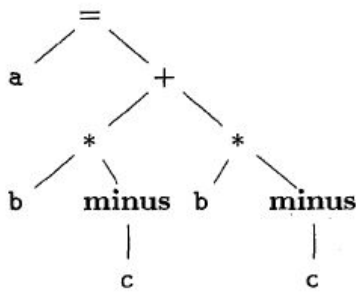
Fig 6.3:DAG **a + a * (b - c) + (b - c) * d**



Syntax tree for **a + a * (b - c) + (b - c) * d**

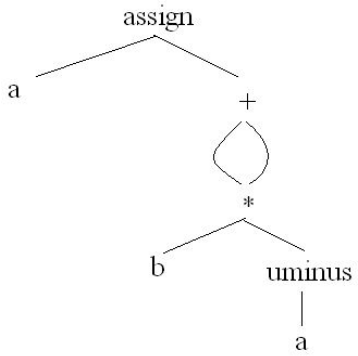


Eg2: syntax tree for assignment statement a=b*-c + b*-c

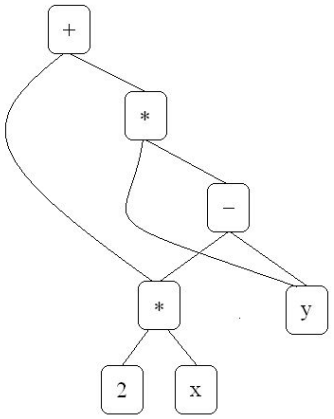


(a) Syntax tree

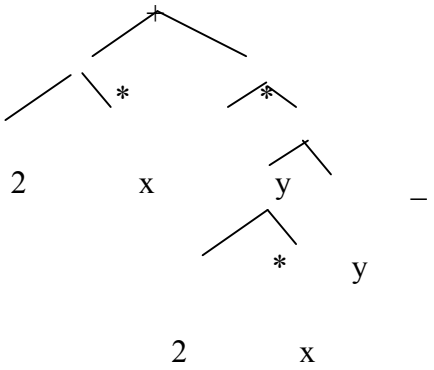
DAG for a=b*-c + b*-c



Eg 3 : For example, the expression 2*x + y * (2*x - y) could be represented by DAG



Syntax tree for 2*x + y * (2*x - y)



The DAG representation may expose instances where redundancies can be eliminated.
SDD to construct DAG for the expression **a + a * (b - c) + (b - c) * d**.

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

1) $p_1 = Leaf(\textbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\textbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\textbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\textbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\textbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\textbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\textbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

6.1.2 The Value-Number Method for Constructing DAG's

In many applications, nodes are implemented as records stored in an array, as in Figure 7.In the figure; each record has a label field that determines the nature of the node. We can refer to a node by its index in the array. The integer index of a node is often called *value number*. For example, using value numbers, we can say node 3 has label +, its left child is node 1, and its right child is node 2.The following algorithm can be used to create nodes for a *dag* representation of an expression.

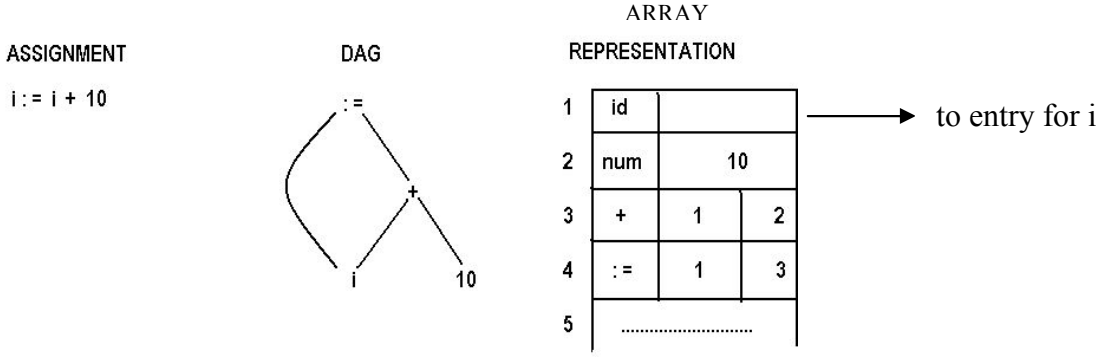


Figure : Nodes o fa DAG for i=i+10 allocated in an array

Algorithm: The value-number method for constructing the nodes of a DAG
Input: Label ***op***, node ***l***, and node ***r***
Output: The value number of a node in the array with signature <***op, l, r***>
Method: Search the array for a node M with label ***op***, left child ***l***, and right child ***r***. If there is such node, return the value number of M. If not, create in the array a new node N with label ***op***, left child ***l***, and right child ***r***, and return its value number.

6.2 Three-Address Code

- In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.

$$x+y*z \Rightarrow t_1 = y * z$$
$$t_2 = x + t_1$$

- ***Example* 6.4**:

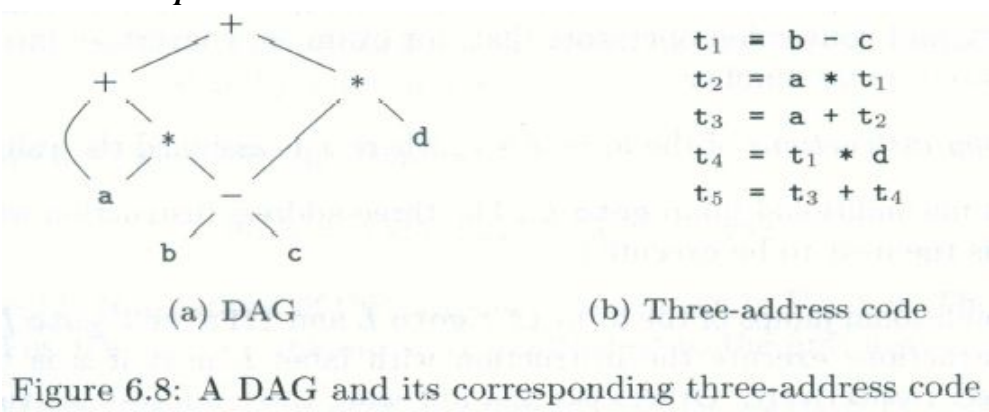


Figure 6.8: A DAG and its corresponding three-address code

Problems: write the 3-address code for the following expression

1. if(x + y * z > x * y +z)
   a=0;
2. (2 + a * (b – c / d)) / e
3. A :=b * -c + b * -c

6.2.1 Address and Instructions

- Three-address code is built from two concepts: addresses and instructions.
- An address can be one of the following:
  - A *name: A source name is replaced by a pointer to its symbol table entry.*
    - **A name:** For convenience, allow source-program names to appear as addresses in three-address code. In an implementation, *a source name is replaced by a pointer to its symbol-table entry*, where all information about the name is kept.
  - A *constant*
    - **A constant:** In practice, a compiler must deal with many different types of constants and variables
  - A *compiler-generated temporary*

- ***A* compiler-generated temporary.** It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

A list of common three-address instruction forms:

Assignment statements

- x= y op z, where op is a binary operation
- x= op y, where op is a unary operation
- Copy statement: x=y
- Indexed assignments: x=y[i] and x[i]=y
- Pointer assignments: x=&y, *x=y and x=*y

Control flow statements

- Unconditional jump: goto L
- Conditional jump: if x relop y goto L ; if x goto L; if False x goto L
- Procedure calls: call procedure p with n parameters and **return y**, is optional

```
param x1
param x2
  …
param xn
call p, n
```

- ***Example* 6.5**:
  - **do i = i +1; while (a[i]<v);**

```
L:   t1 = i + 1          |   100:  t1 = i + 1
     i = t1              |   101:  i = t1
     t2 = i * 8          |   102:  t2 = i * 8
     t3 = a [ t2 ]       |   103:  t3 = a [ t2 ]
     if t3 < v goto L    |   104:  if t3 < v goto 100
```

(a) Symbolic labels. (b) Position numbers.

The multiplication i * 8 is appropriate for an array of elements that each take 8 units of space.

6.2.2 Quadruples

- Three-address instructions can be implemented as objects or as record with fields for the operator and operands.
- Three such representations
  - *Quadruple*, *triples*, and *indirect triples*
- A *quadruple* (or *quad*) has four fields: *op*, arg_1, arg_2, and *result*.
- ***Example* 6.6**:

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

(a) Three-address code

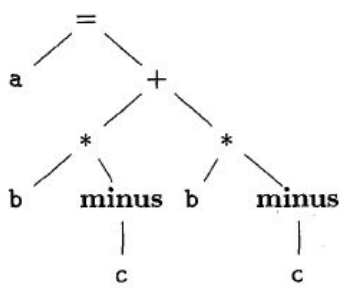
	op	arg_1	arg_2	result
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

6.2.3 Triples

- A *triple* has only three fields: *op*, arg_1, and arg_2
- Using triples, we refer to the result of an operation *x op y* by its position, rather by an explicit temporary name.

***Example* 6.7**



(a) Syntax tree

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

(b) Triples

Fig6.11: Representations of a = b * - *c* + **b** * - c

instruction	
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	...

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

Fig6.12: Indirect triples representation of 3-address code

- The benefit of **Quadruples** over **Triples** can be seen in an optimizing compiler, where instructions are often moved around.
- With ***quadruples***, if we move an instruction that computes a temporary ***t***, then the instructions that use ***t*** require no change. With ***triples***, the result of an operation is referred to by its position, so moving an instruction may require changing all references to that result. ***This problem does not occur with indirect triples***.

6.2.4 Static Single-Assignment Form

- Static single assignment form (SSA) is an intermediate representation that facilitates certain code optimization.
- Two distinct aspects distinguish SSA from three –address code.
  - All assignments in SSA are to variables with distinct names; hence the term *static single-assignment*.

```
p = a + b          p1 = a + b
q = p - c          q1 = p1 - c
p = q * d          p2 = q1 * d
p = e - p          p3 = e - p2
q = p + q          q2 = p3 + q1
```

(a) Three-address code. (b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

```
if (flag) x = -1; else x = 1;
y = x * ;a
```

```
if (flag) x1 = -1; else x2 = 1;
X3 = ϕ(x1, x2)
```

6.3 Types and Declarations

- The applications of types can be grouped under checking and translation:
  - *Type checking* uses logical rules to reason about the behavior of a program at run time. Specifically it ensures that the types of the operands match the type expected by an operator.
  - *Translation Applications*. From the type of a name, a compiler can determine the storage that will be needed for the name at run time.
    - Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic operator, among other things.
- In this section, we examine types and storage layout for names declared within a procedure or a class.

6.3.1 Type Expressions

- Types have structure, which we shall represent using *type expressions*:
  - A type expression is either a basic type or is formed by applying an operator called a *type constructor* or to a type expression
- ***Example* 6.8**:
  - **int[2][3]**
    - "array of 2 arrays of 3 integers each"
    - *array*(2, *array*(3, *integer*))

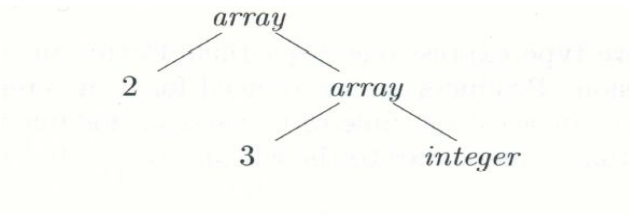


Figure 6.14: Type expression for `int[2][3]`

- Definition of type expressions:
  - A basic type is a type expression.
  - A type name is a type expression.
  - A type expression can be formed by applying the *array* type constructor to a number and a type expression.
  - A record is a data structure with named fields.
  - A type expression can be formed by using the type constructor → for function type.
  - If *s* and *t* are type expressions, then their Cartesian product *s*×*t* is a type expression.
  - Type expressions may contain variables whose values are type expressions.

6.3.2 Type Equivalence

- When type expressions are represented by graphs, two types are *structurally equivalent* if and only if one of the following condition is true:
  - They are the same basic type.
  - They are formed by applying the same constructor to structurally equivalent types.
  - One is a type name that denotes the other.

6.3.3 Declaration

- We shall study types and declarations using a simplified grammar that declares just one name at a time.

$D \rightarrow T$ **id**; $D \mid \varepsilon$

$T \rightarrow B\ C \mid$ **record** '{' D '}'

$B \rightarrow$ **int** | **float**

$C \rightarrow \varepsilon \mid$ [**num**] C

- ***Example* 6.9**: **Storage Layout for Local Names**
  - Computing types and their widths

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

Figure 6.15: Computing types and their widths

- Syntax-directed translation of array types

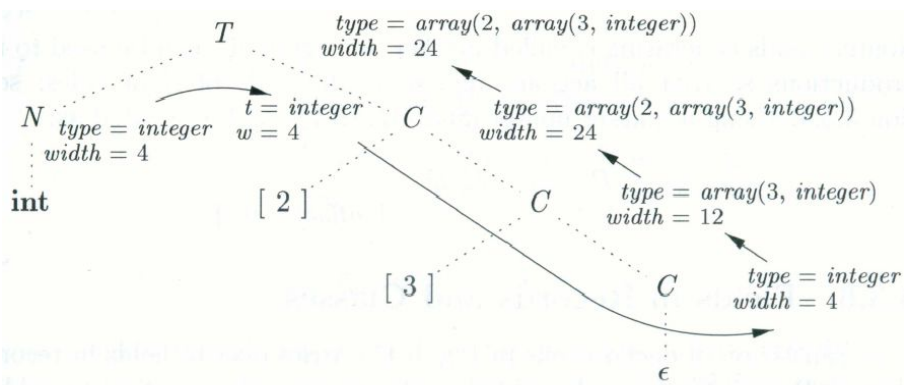


Figure 6.16: Syntax-directed translation of array types

Sequences of Declarations

$P \rightarrow$		$\{ offset = 0; \}$
$\quad D$		
$D \rightarrow T$ **id** ;		$\{ top.put(\mathbf{id}.lexeme,\ T.type,\ offset);$ $offset = offset + T.width; \}$
$\quad D_1$		
$D \rightarrow \epsilon$		

- ***Actions at the end:***

P	$\rightarrow$	$M\ D$	
M	$\rightarrow$	ϵ	$\{ offset = 0; \}$

Fields in Records and Classes

```
float x;
record { float x; float y; } p;
record { int tag; float x; float y; } q;
```

$T \rightarrow$ **record** '{'		$\{ Env.push(top);\ top = \mathbf{new}\ Env();$ $Stack.push(offset);\ offset = 0; \}$
$\quad D$ '}'		$\{ T.type = record(top);\ T.width = offset;$ $top = Env.pop();\ offset = Stack.pop(); \}$

6.4 Translation of Expressions

- An expression with more than one operator, like *a* + *b* * *c*, will translate into instructions with at most one operator per instruction.
- An array reference *A*[*i*][*j*] will expand into a sequence of three-address instructions that calculate an address for the reference.

6.4.1 Operations within Expressions

- ***Example* 6.11**:

  a = b+ -c

  ⬇

  t1 = minus
  t2 = b + t1
  a = t2

Three-address code for expressions

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

Incremental Translation

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

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

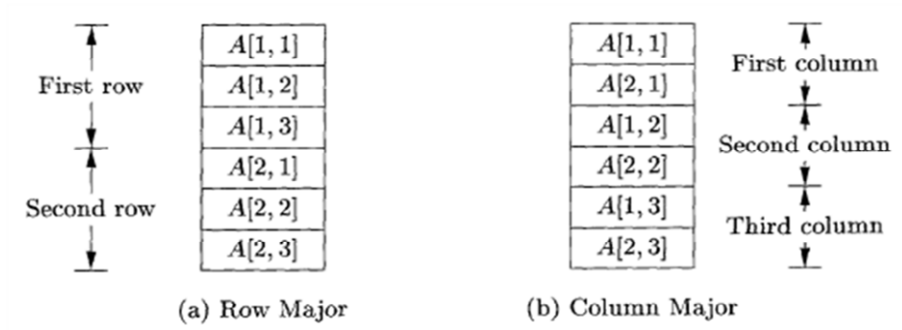
$\mid\ -E_1$ $\{\ E.addr = \textbf{new}\ Temp();$ $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr);\ \}$

$\mid\ (\ E_1\ )$ $\{\ E.addr = E_1.addr;\ \}$

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

Addressing Array Elements

- **Layouts for a two-dimensional array:**



Semantic actions for array reference

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

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

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

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

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

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

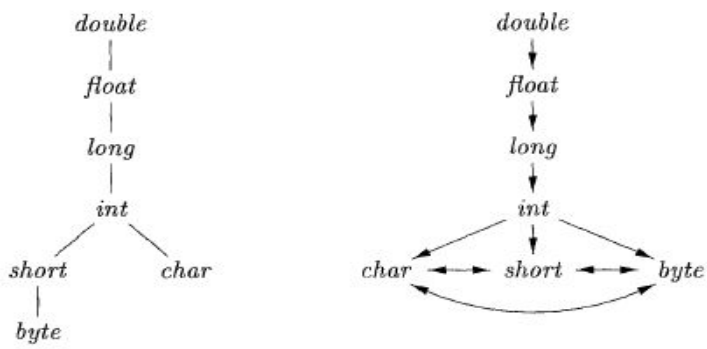
$\mid \; L_1 \; [\; E \;]$ $\{ L.array = L_1.array; \; L.type = L_1.type.elem; \; t = \textbf{new} \; Temp\,(); \; L.addr = \textbf{new} \; Temp\,(); \; gen(t \; '=' \; E.addr \; '*' \; L.type.width); \; gen(L.addr \; '=' \; L_1.addr \; '+' \; t); \}$

Translation of Array References

Nonterminal *L* has three synthesized attributes:

- *L.addr*
- L.array
- L.type

Conversions between primitive types in Java



(a) Widening conversions (b) Narrowing conversions

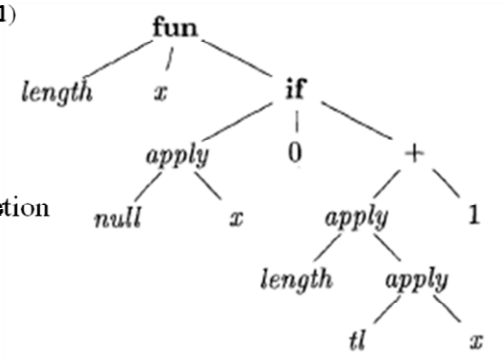
Introducing type conversions into expression evaluation

$$
\begin{aligned}
E \rightarrow E_1 + E_2 \quad \{ & E.type = max(E_1.type, E_2.type); \\
& a_1 = widen(E_1.addr, E_1.type, E.type); \\
& a_2 = widen(E_2.addr, E_2.type, E.type); \\
& E.addr = \textbf{new } Temp\,(); \\
& gen(E.addr\ '='\ a_1\ '+'\ a_2); \}
\end{aligned}
$$

Abstract syntax tree for the function definition

fun length(x) =
if null(x) then 0 else length(tl(x)+1)

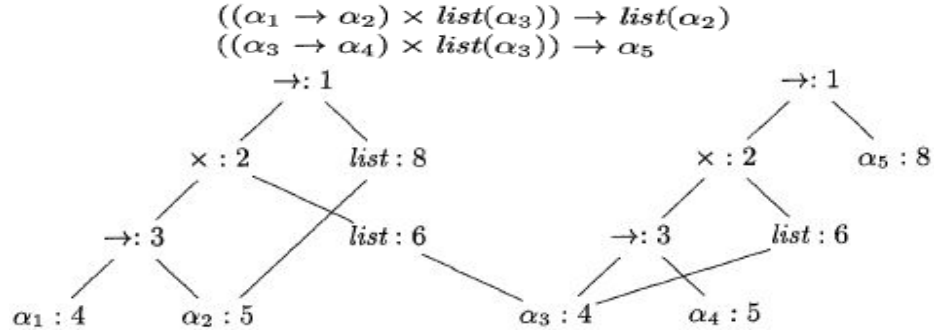
This is a polymorphic function in ML language



Inferring a type for the function *length*

LINE	EXPRESSION	:	TYPE	UNIFY
1)	$length$	:	$\beta \rightarrow \gamma$	
2)	x	:	β	
3)	**if**	:	$boolean \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
4)	$null$	:	$list(\alpha_n) \rightarrow boolean$	
5)	$null(x)$	:	$boolean$	$list(\alpha_n) = \beta$
6)	0	:	$integer$	$\alpha_i = integer$
7)	$+$	:	$integer \times integer \rightarrow integer$	
8)	tl	:	$list(\alpha_t) \rightarrow list(\alpha_t)$	
9)	$tl(x)$	:	$list(\alpha_t)$	$list(\alpha_t) = list(\alpha_n)$
10)	$length(tl(x))$	:	γ	$\gamma = integer$
11)	1	:	$integer$	
12)	$length(tl(x)) + 1$	:	$integer$	
13)	**if**($\cdots$)	:	$integer$	

Algorithm for Unification




```
boolean unify (Node m, Node n)
{
 s = find(m); t = find(n);
 if ( s = t ) return true;
 else if ( nodes s and t represent the same basic type ) return true;
 else if (s is an op-node with children s1 and s2 and
          t is an op-node with children t1 and t2) {
          union(s , t) ;
          return unify(s1, t1) and unify(s2, t2);
 }
 else if s or t represents a variable {
          union(s, t) ;
          return true;
 }
 else return false;
}
```

Control Flow

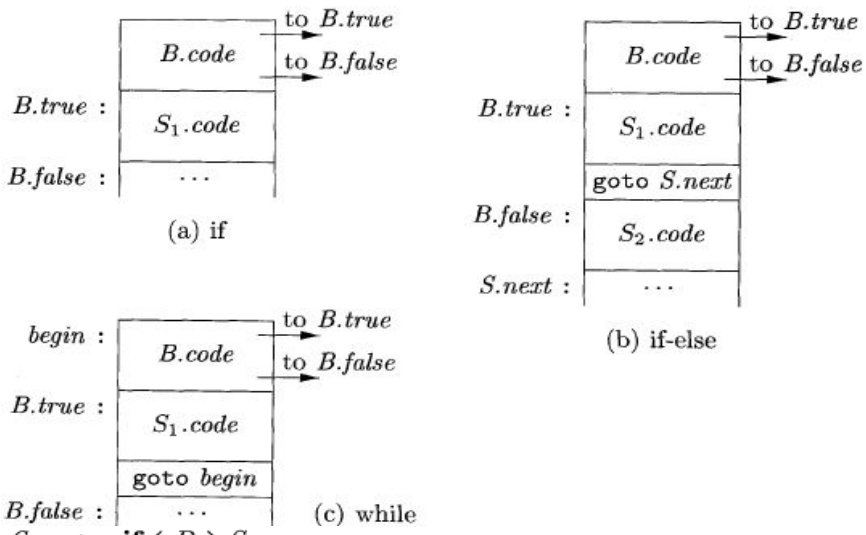
Boolean expressions are often used to:

- *Alter the flow of control.*
- *Compute logical values.*

Short-Circuit Code

```
if ( x < 100 || x > 200 && x != y ) x = 0;
            if x < 100 goto L2
            ifFalse x > 200 goto L1
            ifFalse x != y goto L1
      L2:   x = 0
      L1:
```

Flow-of-Control Statements



$S \rightarrow$ **if** (B) S_1
$S \rightarrow$ **if** (B) S_1 **else** S_2
$S \rightarrow$ **while** (B) S_1

Syntax-directed definition

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ S.next)$ $\mid\mid label(B.false) \mid\mid S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \mid\mid B.code$ $\mid\mid label(B.true) \mid\mid S_1.code$ $\mid\mid gen('\texttt{goto}'\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Generating three-address code for Booleans

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \&\& B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow$ **true**	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow$ **false**	$B.code = gen('\texttt{goto}'\ B.false)$

Translation of a simple if-statement

```
if( x < 100 || x > 200 && x != y ) x = 0;
                 if x < 100 goto L2
                 goto L3
            L3:  if x > 200 goto L4
                 goto L1
            L4:  if x != y goto L2
                 goto L1
            L2:  x = 0
            L1:
```

Backpatching

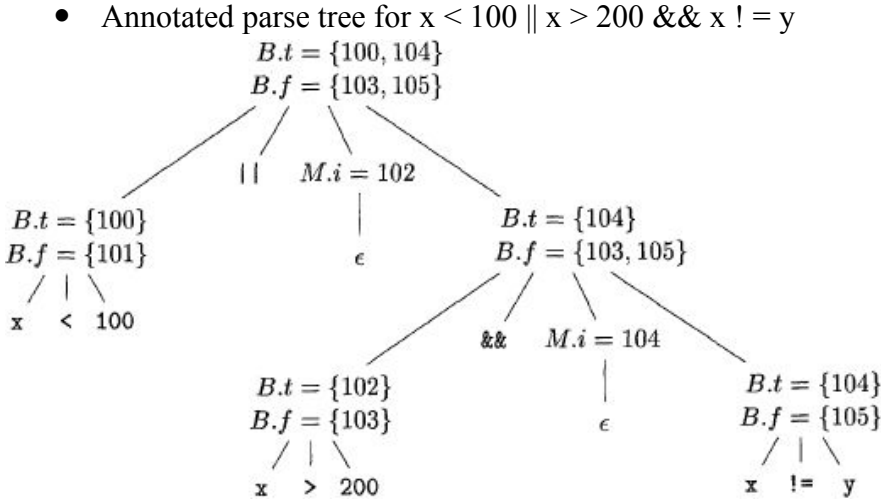
- Previous codes for Boolean expressions insert symbolic labels for jumps
- It therefore needs a separate pass to set them to appropriate addresses
- We can use a technique named backpatching to avoid this
- We assume we save instructions into an array and labels will be indices in the array
- For nonterminal B we use two attributes B.truelist and B.falselist together with following functions:
    - makelist(i): create a new list containing only I, an index into the array of instructions
    - Merge(p1,p2): concatenates the lists pointed by p1 and p2 and returns a pointer to the concatenated list
    - Backpatch(p,i): inserts i as the target label for each of the instruction on the list pointed to by p

Backpatching for Boolean Expressions

$B \rightarrow B_1 \mid\mid M\ B_2 \mid B_1\ \&\&\ M\ B_2 \mid !\ B_1 \mid (\ B_1\ ) \mid E_1\ \mathbf{rel}\ E_2 \mid \mathbf{true} \mid \mathbf{false}$
$M \rightarrow \epsilon$

1)	$B \rightarrow B_1 \mid\mid M\ B_2$	$\{\ backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist;\ \}$
2)	$B \rightarrow B_1\ \&\&\ M\ B_2$	$\{\ backpatch(B_1.truelist, M.instr);$ $B.truelist = B_2.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist);\ \}$
3)	$B \rightarrow\ !\ B_1$	$\{\ B.truelist = B_1.falselist;$ $B.falselist = B_1.truelist;\ \}$
4)	$B \rightarrow (\ B_1\ )$	$\{\ B.truelist = B_1.truelist;$ $B.falselist\ =\ B_1.falselist;\ \}$
5)	$B \rightarrow E_1\ \mathbf{rel}\ E_2$	$\{\ B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $emit('\texttt{if}'\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ '\texttt{goto\ \_}');$ $emit('\texttt{goto\ \_}');\ \}$
6)	$B \rightarrow \mathbf{true}$	$\{\ B.truelist = makelist(nextinstr);$ $emit('\texttt{goto\ \_}');\ \}$
7)	$B \rightarrow \mathbf{false}$	$\{\ B.falselist = makelist(nextinstr);$ $emit('\texttt{goto\ \_}');\ \}$
8)	$M \rightarrow \epsilon$	$\{\ M.instr = nextinstr;\ \}$

- Annotated parse tree for x < 100 || x > 200 && x ! = y



Flow-of-Control Statements

$S\ \rightarrow\ \mathbf{while}\ M_1\ (\ B\ )\ M_2\ S_1$

1) $S \rightarrow$ **if** (B) $M\ S_1$ { $backpatch(B.truelist,\ M.instr)$;
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist)$; }

2) $S \rightarrow$ **if** (B) $M_1\ S_1\ N$ **else** $M_2\ S_2$
{ $backpatch(B.truelist,\ M_1.instr)$;
$backpatch(B.falselist,\ M_2.instr)$;
$temp\ =\ merge(S_1.nextlist,\ N.nextlist)$;
$S.nextlist\ =\ merge(temp,\ S_2.nextlist)$; }

3) $S \rightarrow$ **while** M_1 (B) $M_2\ S_1$
{ $backpatch(S_1.nextlist,\ M_1.instr)$;
$backpatch(B.truelist,\ M_2.instr)$;
$S.nextlist\ =\ B.falselist$;
$emit('\texttt{goto}'\ M_1.instr)$; }

4) $S \rightarrow$ { L } { $S.nextlist\ =\ L.nextlist$; }

5) $S \rightarrow A$; { $S.nextlist\ =$ **null**; }

6) $M \rightarrow \epsilon$ { $M.instr\ =\ nextinstr$; }

7) $N \rightarrow \epsilon$ { $N.nextlist\ =\ makelist(nextinstr)$;
$emit('\texttt{goto\ \_}')$; }

8) $L \rightarrow L_1\ M\ S$ { $backpatch(L_1.nextlist,\ M.instr)$;
$L.nextlist\ =\ S.nextlist$; }

9) $L \rightarrow S$ { $L.nextlist\ =\ S.nextlist$; }

Translation of a switch-statement

switch (E) {
case V_1: S_1
case V_2: S_2
...
case V_{n-1}: S_{n-1}
default: S_n
}

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

```
        code to evaluate E into t
        if t != V1 goto L1
        code for S1
        goto next
L1:     if t != V2 goto L2
        code for S2
        goto next
L2:
        ...
Ln-2:   if t != Vn-1 goto Ln-1
        code for Sn-1
        goto next
Ln-1:   code for Sn
next:
```

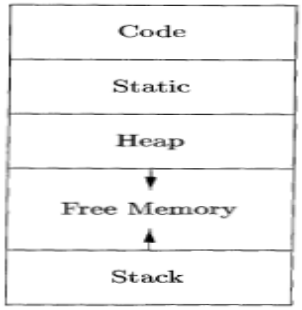
UNIT VII: RUN-TIME ENVIRONMENTS

SYLLABUS

- Storage Organization;
- Stack allocation of space;
- Access to non-local data on the stack;
- Heap management;
- Introduction to garbage collection.

- Compiler must do the storage allocation and provide access to variables and data
- Memory management
  - Stack allocation
  - Heap management
  - Garbage collection

Storage Organization



- Assumes a logical address space
  - Operating system will later map it to physical addresses, decide how to use cache memory, etc.
- Memory typically divided into areas for
  - Program code
  - Other static data storage, including global constants and compiler generated data
  - Stack to support call/return policy for procedures
  - Heap to store data that can outlive a call to a procedure

Static vs. Dynamic Allocation

- Static: Compile time, Dynamic: Runtime allocation
- Many compilers use some combination of following
  - Stack storage: for local variables, parameters and so on
  - Heap storage: Data that may outlive the call to the procedure that created it

Stack allocation is a valid allocation for procedures since procedure calls are nest

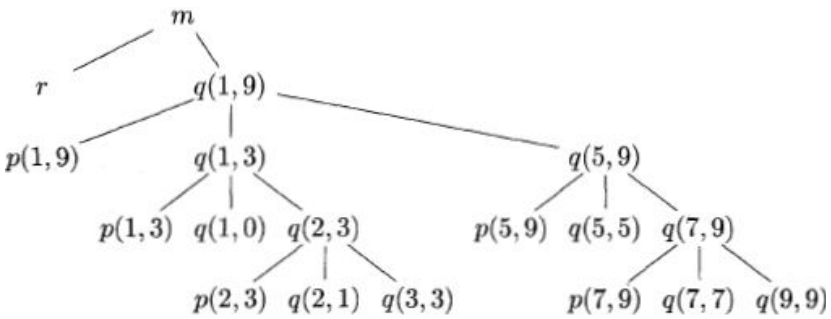
Sketch of a quicksort program

```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
     int i;
     ...
}
int partition(int m, int n) {
     /* Picks a separator value v, and partitions a[m .. n] so that
        a[m .. p - 1] are less than v, a[p] = v, and a[p + 1 .. n] are
        equal to or greater than v. Returns p. */
     ...
}
void quicksort(int m, int n) {
     int i;
     if (n > m) {
          i = partition(m, n);
          quicksort(m, i-1);
          quicksort(i+1, n);
     }
}
main() {
     readArray();
     a[0] = -9999;
     a[10] = 9999;
     quicksort(1,9);
}
```

Activation for Quicksort

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

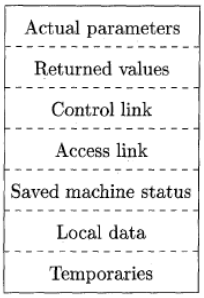
Activation tree representing calls during an execution of quicksort



Activation records

- Procedure calls and returns are usaully managed by a run-time stack called the control stack.
- Each live activation has an activation record (sometimes called a frame)
- The root of activation tree is at the bottom of the stack
- The current execution path specifies the content of the stack with the last activation has record in the top of the stack.

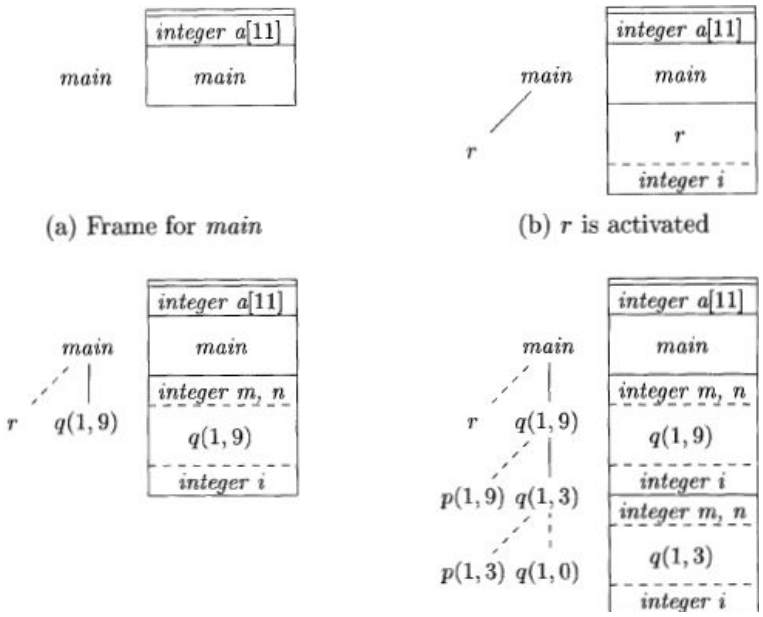
A General Activation Record



Activation Record

- Temporary values
- Local data
- A saved machine status
- An "access link"
- A control link
- Space for the return value of the called function
- The actual parameters used by the calling procedure
- Elements in the activation record:
  - temporary values that could not fit into registers
  - local variables of the procedure
  - saved machine status for point at which this procedure called. includes return address and contents of registers to be restored.
  - access link to activation record of previous block or procedure in lexical scope chain
  - control link pointing to the activation record of the caller
  - space for the return value of the function, if any
  - actual parameters (or they may be placed in registers, if possible)

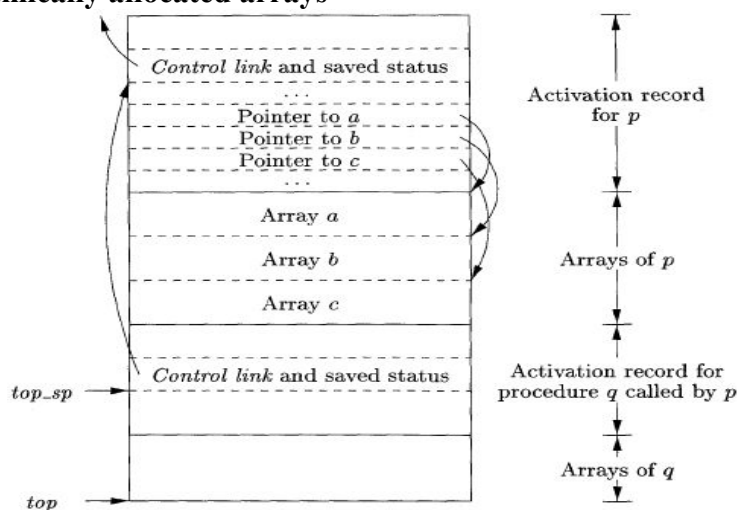
Downward-growing stack of activation records



Designing Calling Sequences

- Values communicated between caller and callee are generally placed at the beginning of callee's activation record
- Fixed-length items: are generally placed at the middle
- Items whose size may not be known early enough: are placed at the end of activation record
- We must locate the top-of-stack pointer judiciously: a common approach is to have it point to the end of fixed length fields.

Access to dynamically allocated arrays



- **ML**
- ML is a functional language
- Variables are defined, and have their unchangeable values initialized, by a statement of the form:

val (name) = (expression)

- Functions are defined using the syntax:

fun (name) ((arguments)) = (body)

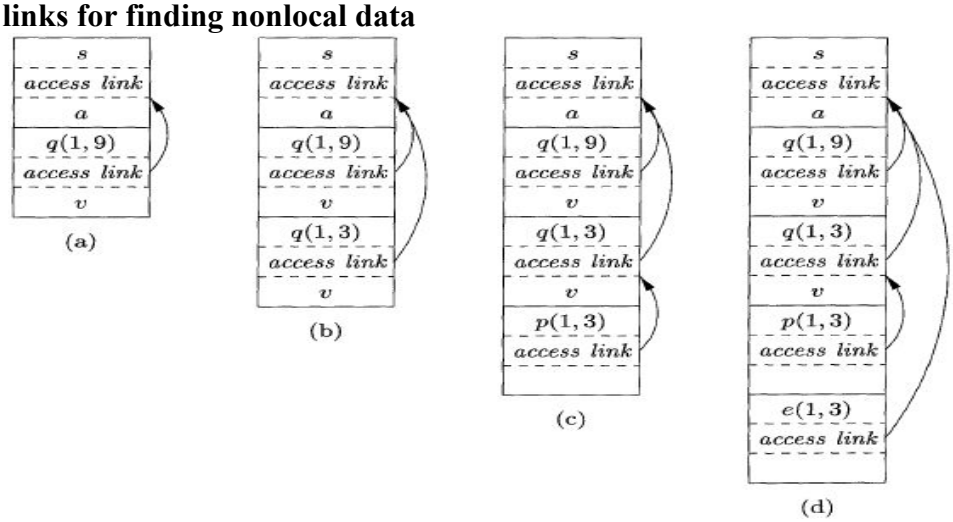
- For function bodies we shall use let-statements of the form:

let (list of definitions) in (statements) end

A version of quicksort, in ML style, using nested functions

```
fun sort(inputFile, outputFile) =
    let
        val a = array(11,0);
        fun readArray(inputFile) = ··· ;
              ··· a ··· ;
        fun exchange(i,j) =
              ··· a ··· ;
        fun quicksort(m,n) =
            let
                val v = ··· ;
                fun partition(y,z) =
                     ··· a ··· v ··· exchange ···
            in
                ··· a ··· v ··· partition ··· quicksort
            end
    in
        ··· a ··· readArray ··· quicksort ···
    end;
```

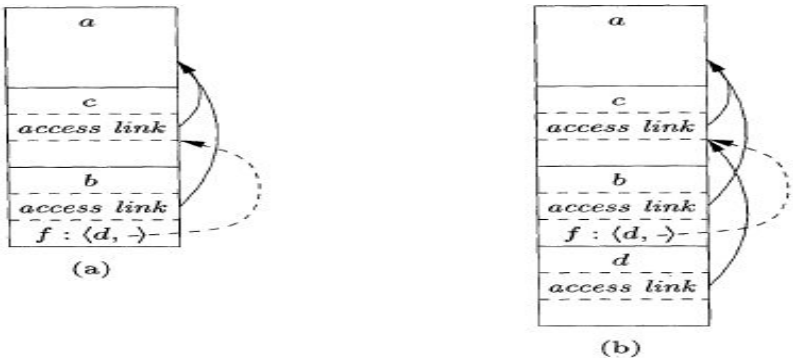
Access links for finding nonlocal data



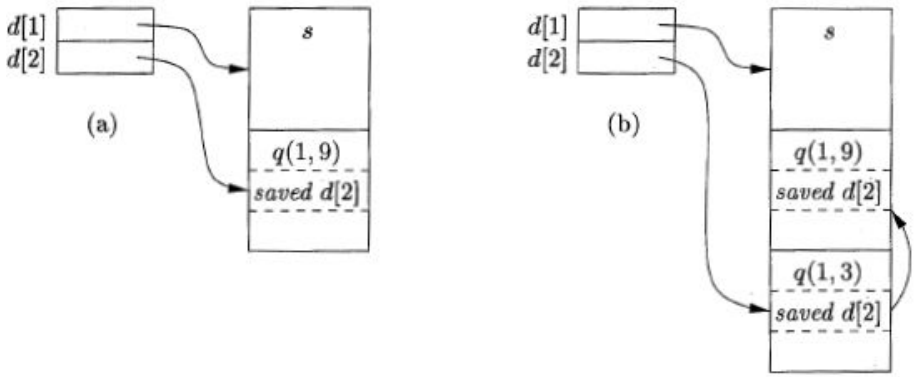
Sketch of ML program that uses function-parameters

```
fun a(x) =
    let
        fun b(f) =
            ··· f ··· ;
        fun c(y) =
            let
                fun d(z) = ···
            in
                ··· b(d) ···
            end
    in
        ··· c(1) ···
    end;
```

Actual parameters carry their access link with them



Maintaining the Display



Memory Manager

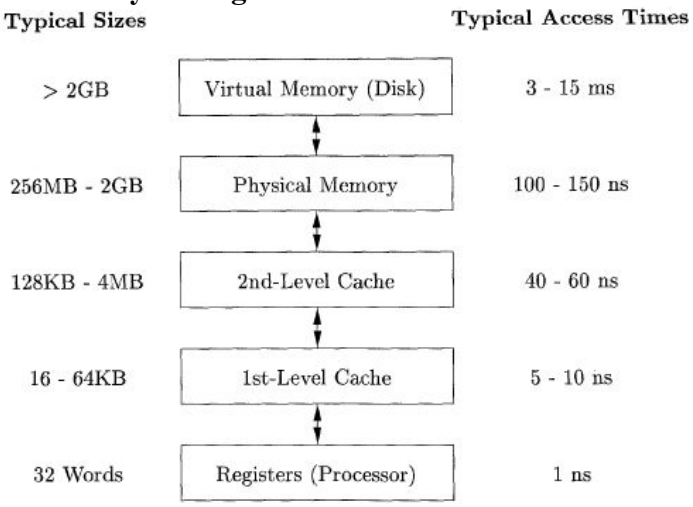
Two basic functions:

- Allocation
- Deallocation

- Properties of memory managers:
  - Space efficiency
  - Program efficiency

Low overhead

Typical Memory Hierarchy Configurations



Locality in Programs

The conventional wisdom is that programs spend 90% of their time executing 10% of the code:

- Programs often contain many instructions that are never executed.
- Only a small fraction of the code that could be invoked is actually executed in a typical run of the program.
- The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.

UNIT-VIII: CODE GENERATION

SYLLABUS

- Issues in the design of Code Generator;
- The Target Language;
- Addresses in the target code;
- Basic blocks and Flow graphs;
- Optimization of basic blocks;
- A Simple Code Generator

- **The final phase in our compiler model**

Figure 8.1: Position of code generator

- Requirements imposed on a code generator
  - Preserving the semantic meaning of the source program and being of high quality
  - Making effective use of the available resources of the target machine
  - The code generator itself must run efficiently.
- A code generator has three primary tasks:
  - Instruction selection, register allocation, and instruction ordering

Issue in the Design of a Code Generator

- General tasks in almost all code generators: instruction selection, register allocation and assignment.
  - The details are also dependent on the specifics of the intermediate representation, the target language, and the run-tie system.
- The most important criterion for a code generator is that it produce correct code.

Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design

Input to the Code Generator

- The input to the code generator is
  - the intermediate representation of the source program produced by the frontend along with
  - information in the symbol table that is used to determine the run-time address of the data objects denoted by the names in the IR.
- Choices for the IR
  - Three-address representations: quadruples, triples, indirect triples
  - Virtual machine representations such as bytecodes and stack-machine code
  - Linear representations such as postfix notation
  - Graphical representation such as syntax trees and DAG's
- Assumptions
  - Relatively lower level IR
  - All syntactic and semantic errors are detected.

The Target Program

- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code.
- The most common target-machine architecture are RISC, CISC, and stack based.

- A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
- A CISC machine typically has few registers, two-address instructions, and variety of addressing modes, several register classes, variable-length instructions, and instruction with side effects.

In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack

- Java Virtual Machine (JVM)
  - Just-in-time Java compiler
- Producing the target program as
  - An absolute machine-language program
  - Relocatable machine-language program
  - An assembly-language program
- In this chapter
  - Use very simple RISC-like computer as the target machine.
  - Add some CISC-like addressing modes
  - Use assembly code as the target language.

Instruction Selection

- The code generator must *map* the IR program into a code sequence that can be executed by the target machine.
- The complexity of the mapping is determined by the factors such as
  - The level of the IR
  - The nature of the instruction-set architecture
  - The desired quality of the generated code
- If the IR is high level, use code templates to translate each IR statement into a sequence of machine instruction.
  - Produces poor code, needs further optimization.
- If the IR reflects some of the low-level details of the underlying machine, then it can use this information to generate more efficient code sequence.

Instruction Selection

- The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For example,
  - The uniformity and completeness of the instruction set are important factors.
  - Instruction speeds and machine idioms are another important factor.
    - If we do not care about the efficiency of the target program, instruction selection is straightforward.

```
x = y + z ⇒   LD   R0, y
              ADD  R0, R0, z
              ST   x, R0

a = b + c ⇒   LD   R0, b
d = a + e     ADD  R0, R0, c
              ST   a, R0
              LD   R0, a      Redundant
              ADD  R0, R0,e
              ST   d, R0
```

- The quality of the generated code is usually determined by its speed and size.
- A given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations.
- A naïve translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.
- For example use **INC** for **a=a+1** instead of

  LD R0,a
  ADD R0, R0, #1
  ST a, R0
- We need to know instruction costs in order to design good code sequences but, unfortunately, accurate cost information is often difficult to obtain.

Register Allocation

- A key problem in code generation is deciding what values to hold in what registers.
- Efficient utilization is particularly important.
- The use of registers is often subdivided into two subproblems:
  1. ***Register Allocation***, during which we select the set of variables that will reside in registers at each point in the program.
  2. ***Register assignment***, during which we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers to variables is difficult, even with single-register machine.
- Mathematically, the problem is NP-complete.

Register pairs (even/odd numbered) for some operands & results

- Multiplication instruction is in the form M x, y where x, the multiplicand, is the even register of even/odd register pair and y, the multiplier, is the odd register.
- The product is occupies the entire even/odd register pair.
- D x, y where the dividend occupies an even/odd register pair whose even register is x, the divisor is y. After division, the even register holds the remainder and the **odd register the quotient.**

Example:

```
t = a + b
t = t * c
t = t / d
```

(a)

```
t = a + b
t = t + c
t = t / d
```

(b)

Figure 8.2: Two three-address code sequences

```
L   R1,a
A   R1,b
M   R0,c
D   R0,d
ST  R1,t
```

(a)

```
L    R0, a
A    R0, b
A    R0, c
SRDA R0, 32
D    R0, d
ST   R1, t
```

(b)

Figure 8.3: Optimal machine-code sequences

Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- However, picking a best order in the general case is a difficult NP-complete problem.

A Simple Target Machine Model

- Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps.
- The underlying computer is a byte-addressable machine with *n* general-purpose registers.
- Assume the following kinds of instructions are available:
  - Load operations Ex: LD dst, addr
  - Store operations Instruction like ST x, r
  - Computation operations of the form OP dst,src1,src2
  - Unconditional jumps : The instruction BR L
  - Conditional jumps : Bcond r, L
  - Ex: BLTZ r, L

A Simple Target Machine Model

- *Example* **8.2**:

```
x = y –z ⇒ LD  R1, y
           LD  R2, z
           SUB R1, R1, R2
           ST  x, R1

b = a[i] ⇒ LD  R1, i
           MUL R1, R1, 8
           LD  R2, a(R1)
           ST  b, R2

a[j] = c ⇒ LD  R1, c
           LD  R2, j
           MUL R2, R2, 8
           ST  a(R2), R1

x = *p ⇒   LD  R1, p
           LD  R2, 0(R1)
           ST  x, R2

*p = y ⇒   LD  R1, p
           LD  R2, y
           ST  0(R1), R2

if x < y goto L ⇒ LD   R1, x
                  LD   R2, y
                  SUB  R1, R1, R2
                  BLTZ R1, L
```

Program and Instruction Costs

- For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands.
- Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost f one.
- For example,
  - **LD R0, R1** cost = 1
  - **LD R0, M** cost = 2
  - **LD R1, *100(R2)** cost = 3

Addresses in the Target Code

- We show how names in the IR can be converted into addresses in the target code by looking at code generation for simple procedure calls and returns using static and stack allocation.
- In Section 7.1, we described how each executing program runs in its own logical address space that was partitioned into four code and data areas:
  1. A statically determined area *Code* that holds the executable target code.
  2. A statically determined data area *Static*, for holding global constants and other data generated by the compiler.
  3. A dynamically managed area *Heap* for holding data objects that are allocated and freed during program execution.

A dynamically managed area *Stack* for holding activation records as they are created and destroyed during procedure calls and returns

Target Code Addresses

- Four areas of memory: Code, Static, Heap and Stack
- Can use one static base location for Code and Static variable area
  - procedures have a location (offset) of the code in this area
  - global variables allocated in the static area also given offsets here
- Other program variables, local variables and formal parameters, are given offset locations with regard to a stack activation record pointer

Basic Blocks and Flow Graphs

- Representation of intermediate code as a graph
  - nodes of the graph are basic blocks, where the flow of control can only enter at the first instruction and leave through the last
  - edges indicate which blocks can follow other blocks, representing the jumps in the code
- Useful for discussing code generation
- Defining basic blocks
  - separate sequence of TAC (three address code) into basic blocks by identifying the first instruction as a leader:
    - very first instruction is a leader
    - any instruction that is the target of a jump is a leader
    - any instruction following a jump is a leader

Basic Blocks

- A basic block is a sequence of statements such that:
  - Flow of control enters at the beginning of the basic block
  - Flow of control leaves at the end of the basic block
  - No possibility of halting or branching except at end
- A name is *live* at a given point if its value will be used again in the program
- Each basic block has a first statement known as the *leader* of the basic block

Partitioning Code into Basic Blocks

- Algorithm must determine all leaders:
  - The first statement is a leader
  - Any statement that is the target of a conditional or unconditional goto is a leader
  - Any statement immediately following a goto or unconditional goto is a leader
- A basic block:
  - Starts with a leader
  - Includes all statements up to but not including the next leader

Example of Basic blocks

psuedo code to initialize a 10 by 10 array to be the identity matrix,

```
for i from 1 to 10 do
   for j from 1 to 10 do
         a[i, j] = 0.0
for i from 1 to 10 do
         a[i, j] = 1.0
```

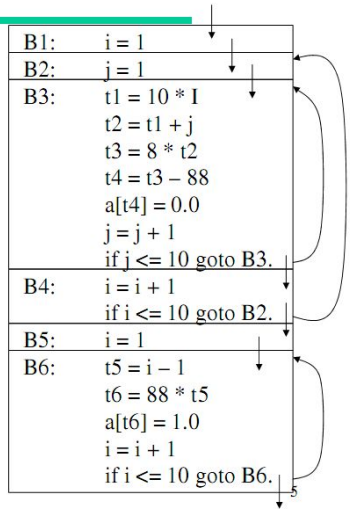
Three address code, assuming a is the starting address of the array in row-major form and that each element takes 8 bytes each:

```
1.  i = 1                    leader
2.  j = 1                    leader
3.  t1 = 10 * i              leader
4.  t2 = t1 + j
5.  t3 = 8 * t2
6.  t4 = t3 – 88
7.  a[t4] = 0.0
8.  j = j + 1
9.  if j <= 10 goto 3.
10. i = i + 1                leader
11. if i <= 10 goto 2.
12. i = 1                    leader
13. t5 = i – 1               leader
14. t6 = 88 * t5
15. a[t6] = 1.0
16. i = i + 1
17. if i <= 10 goto 13.
```

4

Graph Representation

- Basic blocks connected by edges representing jumps
- add an entry and exit point
- can also identify set of nodes as a loop
  - loop entry is only node with predecessor outside the loop
  - every node in the loop has a path to the entry



5

Good Code Generation for Basic Blocks

- Some local optimization can be achieved by building a DAG representation of the basic block
  - node for each instruction in the block whose children are the previous statements giving the last definition of the operands
- Eliminate local common subexpressions
  - check if there are nodes with the same operator and the same children
- Dead code elimination
  - if there is a node with no ancestors and with no live variables, then that node can be eliminated
- Algebraic identities
  - arithmetic identities (e.g. $x - 0 = x = x + 0 = 0 + x \ldots$)
  - reduction in strength, replacing expensive operations with cheaper ones (e.g. $x^2 = x * x$, $2 * x = x + x$)
  - constant folding - evaluate constant expressions at compile time (e.g. 2 * 3.14)

Simple code generation

- Generates code considering just one basic block, but introduces the ideas of register allocation
- assume that we keep information about registers
  - register descriptor keeps track of which variable names have a current value in that register
  - address descriptor for each program variable keeps track of where the current value of the variable can be found
- GetRegister function gets an appropriate register for any operand of the TAC
- For the instruction x = y + z
  - call GetRegister for each of the operands
    - if y is not already in its register: lw Ry, y
    - if z is not already in its register: lw Rz, z
    - give the instruction add Rx, Ry, Rz

Simple generation updates descriptors

- At end of basic block, ignore TAC temporaries (not live)
  - for each program variable x, if current value of x is not in memory, issue a sw instruction
- Updating descriptors:
  - For the instruction lw Ry, y, change register descr. for Ry to only hold y
  - add Ry to address descriptor of y as a location
  - for the instruction sw x, Rx, change the address descr of x to include memory address of x
  - for each operation add Rx, Ry, Rz
    - register descr Rx holds only x
    - address descr of x has only Rx (not any memory location)
    - remove Rx from address descr of any other variable

GetRegister function

- Pick a register Ry for any variable y that is an operand
  - if y is already in a register, pick it (no load instruction needed)
  - if y is not in a register, but one is free, pick that register (and load it)
  - if y is not in a register but there is not register free, consider candidate registers R
    - any register with a variable v, whose descriptor says its current value is in memory already
    - any register with the variable x, the result of the instruction, and x is not also one of the operands (x will be rewritten anyway)
    - any register with a variable v that is not used later
    - otherwise, generate a store instruction sw R, v to "spill" v
    - repeat these steps for other variables in the register, and pick a register with the fewest number of stores
- Pick a register Rx for the variable x that is the result
    - in addition to above: any register holding only x
    - if y is not used later, use Ry to hold the result Rx (e.g. x = y)

- Code optimization:
  - A transformation to a program to make it run faster and/or take up less space
  - Optimization should be safe, preserve the meaning of a program.
  - Code optimization is an important component in a compiler

  - Examples:
    - Flow of control optimization:

goto L1 … L1: goto L2	goto L2 … L1: goto L2

if a < b goto L1 … L1: goto L2	if a<b goto L2 … L1: goto L2

goto L1 … L1: if a < b goto L2	if a < b goto L2 goto L3 … L1: L3:

- Algebraic simplification:
  x : = x+0
  x := x*1 ==➔ nop
- Reduction in strength
  X^2 → x * x
  X * 4 → x << 2
- Instruction selection

Sometimes some hardware instructions can implement certain operation efficiently.

- Code optimization can either be high level or low level:
  - High level code optimizations:

        - Loop unrolling, loop fusion, procedure inlining
    - Low level code optimizations:
        - Instruction selection, register allocation
    - Some optimization can be done in both levels:
        - Common subexpression elimination, strength reduction, etc.
    - Flow graph is a common intermediate representation for code optimization.
- Code optimization can either be high level or low level:
    - High level code optimizations:
        - Loop unrolling, loop fusion, procedure inlining
    - Low level code optimizations:
        - Instruction selection, register allocation
    - Some optimization can be done in both levels:
        - Common subexpression elimination, strength reduction, etc.
    - Flow graph is a common intermediate representation for code optimization.
- Basic block: a sequence of consecutive statements with exactly 1 entry and 1 exit.
- Flow graph: a directed graph where the nodes are basic blocks and block B1→ block B2 if and only if B2 can be executed immediately after B1:
- Algorithm to construct flow graph:
    - Finding leaders of the basic blocks:
        - The first statement is a leader
        - Any statement that is the target of a conditional or unconditional goto is a leader
        - Any statement that immediately follows a goto or conditional goto statement is a leader
    - For each leader, its basic block consists all statements up to the next leader.
    - B1→B2 if and only if B2 can be executed immediately after B1.
- Example:

```
100:  sum = 0
101:   j = 0
102:  goto  107
103:  t1 = j << 2
104:  t2 = addr(a)
105:  t3 = t2[t1]
106:  sum = sum + t3
107:  if j < n goto 103
```

- Optimizations within a basic block is called local optimization.
- Optimizations across basic blocks is called global optimization.
- Some common optimizations:
    - Instruction selection

- Register allocation
- Common subexpression elimination
- Code motion
- Strength reduction
- Induction variable elimination
- Dead code elimination
- Branch chaining
- Jump elimination
- Instruction scheduling
- Procedure inlining
- Loop unrolling
- Loop fusing
- Code hoisting

- Instruction selection:
  - Using a more efficient instruction to replace a sequence of instructions (space and speed).
  - Example:

Mov R2, (R3)
Add R2, #1, R2
Mov (R3), R2 ➔ Add (R3), 1, (R3)

- Register allocation: allocate variables to registers (speed)
- Example:

```
    M[R13+sum] = 0
    M[R13+j] = 0
    GOTO L18
L19:
    R0 = M[R13+j] * 4
    M[R13+sum] = M[R13+sum]
                 +M[R0+_a]
    M[R13+j] = M[R13+j]+1
L18:
    NZ = M[R13+j] - M[_n]
    if NZ < 0 goto L19
```

```
    R2 = 0
    R1 = 0
    GOTO L18
L19:
    R0 = R1 * 4
    R2 = R2+M[R0+_a]
    R1 = R1+1
L18:
    NZ = R1 - M[_n]
    if NZ < 0 goto L19
```

- Code motion: move a loop invariant computation before the loop
- Example:

```
    R2 = 0
    R1 = 0
    GOTO L18
L19:
    R0 = R1 * 4
    R2 = R2+M[R0+_a]
    R1 = R1+1
L18:
    R4 = M[_n]
    NZ = R1 – R4
    if NZ < 0 goto L19
```

```
    R2 = 0
    R1 = 0
    R4 = M[_n]
    GOTO L18
L19:
    R0 = R1 * 4
    R2 = R2+M[R0+_a]
    R1 = R1+1
L18:
    NZ = R1 – R4
    if NZ < 0 goto L19
```

- **Strength reduction: replace expensive operation by equivalent cheaper operations**
- **Example:**

```
      R2 = 0
      R1 = 0
      R4 = M[_n]
      GOTO L18
L19:
      R0 = R1 * 4
      R2 = R2+M[R0+_a]
      R1 = R1+1
L18:
      NZ = R1 – R4
      if NZ < 0 goto L19
```

```
      R2 = 0
      R1 = 0
      R4 = M[_n]
      R3 = _a
      GOTO L18
L19:
      R2 = R2+M[R3]
      R3 = R3 + 4
      R1 = R1+1
L18:
      NZ = R1 – R4
      if NZ < 0 goto L19
```

- Induction variable elimination: can induce value from another variable.

Example:

```
      R2 = 0
      R1 = 0
      R4 = M[_n]
      R3 = _a
      GOTO L18
L19:
      R2 = R2+M[R3]
      R3 = R3 + 4
      R1 = R1+1
L18:
      NZ = R1 – R4
      if NZ < 0 goto L19
```

```
      R2 = 0
      R4 = M[_n] << 2
      R3 = _a
      GOTO L18
L19:
      R2 = R2+M[R3]
      R3 = R3 + 4
L18:
      NZ = R3 – R4
      if NZ < 0 goto L19
```

- Common sub-expression elimination: an expression was previously calculated and the variables in the expression have not changed. Can avoid re-computing the expression.
- **Example:**

```
R1 = M[R13+I] << 2
R1 = M[R1+_b]
R2 = M[R13+I] << 2;
R2 = M[R2+_b]
```

```
R1=M[R13+I] << 2
R1 = M[R1+_b]
R2 = R1
```

ALGEBRAIC TRANSFORMATION

Countless algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set. The useful ones are those that simplify expressions or replace expensive operations by cheaper ones. For example, statements
such as

x := x +0

Or

x := x*1

can be eliminated from a basic block without changing the set of expressions it computes. The exponentiation operator in the statements

x := y ** 2

usually requires a function call to implement. Using an algebraic transformation, this statement can be replaced by cheaper, but equivalent statement

x := y*y

FLOW GRAPHS

We can add the flow-of –control information to the set of basic blocks making up a program by constructing a directed graph called a flow graph. The nodes of the flow graph are the basic blocks. One node is distinguished as initial; it is the block whose leader is the first statement. There is a directed edge from block B1 to block B2can be immediately follow B1in some execution sequence; that is, if

1. there is a conditional or unconditional jump from the last statement of B2, or
2. B2 immediately follow B1in the order of the program, and B1 does not end in the unconditional jump

B1 is a predecessor of B2, and B2is a successor of B1.

Example 4:The flow graph of the program of fig. 7 is shown in fig. 9, B1 is the initial node.

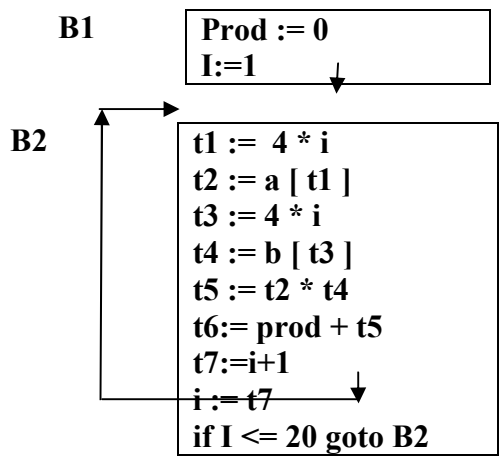


Fig .9 flow graph for program

REPRESENTATION OF BASIC BLOCKS

Basic Blocks are represented by variety of data structures. For example, after partitioning the three address statements by Algorithm 1, each basic block can be represented by a record consisting of a count of number of quadruples in the block, followed by a pointer to the leader of the block, and by the list of predecessors and successors of the block. For example the block B2 running from the statement (3) through (12) in the intermediate code of figure 2 were moved elsewhere in the quadruples array or were shrunk, the (3) in if i<=20 goto(3) would have to be changed.

LOOPS

Loop is a collection of nodes in a flow graph such that

1. All nodes in the collection are *strongly connected*; from any node in the loop to any other, there is path of length one or more, wholly within the loop, and

2. The collection of nodes has a unique *entry*, a node in the loop such that is, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

A loop that contains no other loops is called an *inner loop*.

REDUNTANT LOADS AND STORES

If we see the instructions sequence

(1) MOV R0,a
(2) MOV a,R0

-we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

UNREACHABLE CODE

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1.In C, the source code might look like:

```
#define debug 0
    ....
If ( debug  ) {
  Print debugging information
}
```

In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L2
Goto L2
L1: print debugging information
L2:                                    ……(a)
```

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**, (a) can be replaced by:

```
If debug ≠1 goto L2
Print debugging information
L2:                                    ……(b)
```

As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

```
If debug ≠0 goto L2
Print debugging information
L2:                                    ……(c)
```

As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

FLOW-OF-CONTROL OPTIMIZATIONS

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

```
goto L2
```

….
L1 : gotoL2

by the sequence

goto L2
….
L1 : goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

if a < b goto L1
….
L1 : goto L2

can be replaced by

if a < b goto L2
….
L1 : goto L2

Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1
……..
L1:if a<b goto L2
L3: ……..(1)

may be replaced by

if a<b goto L2
goto L3
…….
L3: …….(2)

While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

ALGEBRAIC SIMPLIFICATION

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them .For example, statements such as

x := x+0

Or

x := x * 1

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

ELIMINATION OF COMMON SUBEXPRESSIONS

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where its referenced when encountered again – of course providing the variable values in the expression still remain constant.

ELIMINATION OF DEAD CODE

Its possible that a large amount of dead(useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of

construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code

REDUCTION IN STRENGTH

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

USE OF MACHINE IDIOMS

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and

auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : =i+1.

Getting Better Performance

Dramatic improvements in the running time of a program-such as cutting the running time form a few hours to a few seconds-are usually obtained by improving the program at all levels, from the source level to the target level, as suggested by fig. At each level, the available options fall between the two extremes of finding a better algorithm and of implementing a given algorithm so that fewer operations are performed.

Algorithmic transformations occasionally produce spectacular improvements in running time. For example, Bentley relates that the running time of a program for sorting N elements dropped from 2.02N^2 microseconds to 12Nlog2N microseconds then a carefully coded "insertion sort" was replaced by "quicksort".

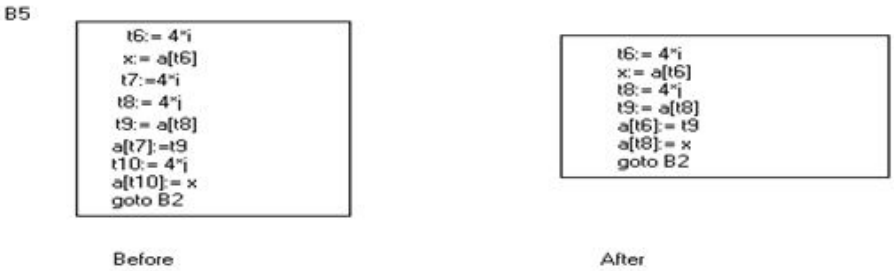
THE PRINCIPAL SOURCES OF OPTIMIZATION

Here we introduce some of the most useful code-improving transformations. Techniques for implementing these transformations are presented in subsequent sections. A transformation of a program is called local if it can be performed by looking only at the statements in a bas9ic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes. Common subexpression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of these duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language. For example, block B5 shown in fig recalculates 4*i and 4*j.

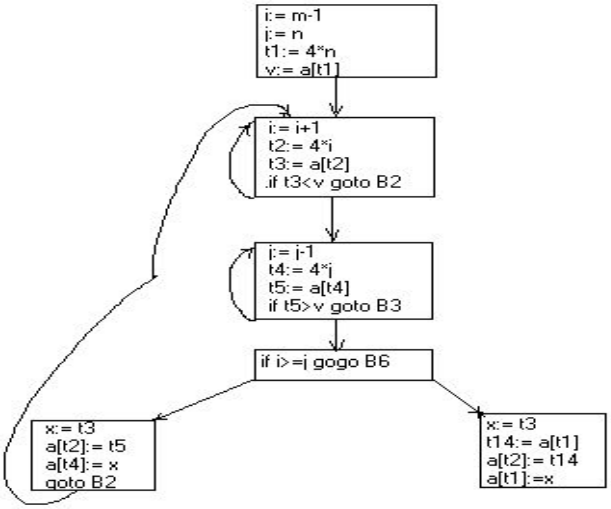


Local common subexpression elimination

Common Subexpressions

An occurrence of an expression E is called a common subexpression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value. For example, the assignments to t7 and t10 have the common subexpressions 4*I and 4*j, respectively, on the right side in Fig. They have been eliminated in Fig by using t6 instead of t7 and t8 instead of t10. This change is what would result if we reconstructed the intermediate code from the dag for the basic block.

Example: Fig shows the result of eliminating both global and local common subexpressions from blocks B5 and B6 in the flow graph of Fig. We first discuss the transformation of B5 and then mention some subtleties involving arrays.



B5 and B6 after common sub-expression elimination

After local common sub-expressions are eliminated B5 still evaluates 4*i and 4*j, as shown in the earlier fig. Both are common sub-expressions; in particular, the three statements

t8:= 4*j; t9:= a[t[8]; a[t8]:=x

in B5 can be replaced by

t9:= a[t4]; a[t4:= x using t4 computed in block B3. In Fig. observe that as control passes from the evaluation of 4*j in B3 to B5, there is no change in j, so t4 can be used if 4*j is needed.

Another common sub-expression comes to light in B5 after t4 replaces t8. The new expression a[t4] corresponds to the value of a[j] at the source level. Not only does j retain its value as control leaves b3 and then enters B5, but a[j], a value computed into a temporary t5, does too because there are no assignments to elements of the array a in the interim. The statement

t9:= a[t4]; a[t6]:= t9

in B5 can therefore be replaced by

a[t6]:= t5

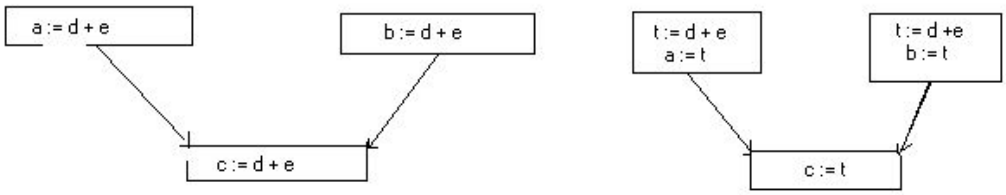
The expression in blocks B1 and B6 is not considered a common subexpression although t1 can be used in both places.After control leaves B1 and before it reaches B6,it can go through B5,where there are assignments to a.Hence, a[t1] may not have the same value on reaching B6 as it did in leaving B1, and it is not safe to treat a[t1] as a common subexpression.

Copy Propagation

Block B5 in Fig. can be further improved by eliminating x using two new transformations. One concerns assignments of the form f:=g called copy statements, or copies for short. Had we gone into more detail in Example 10.2, copies would have arisen much sooner, because the algorithm for eliminating common subexpressions introduces them, as do several other algorithms. For example, when the common subexpression in c:=d+e is eliminated in Fig., the algorithm uses a new variable t to hold the value of d+e. Since control may reach c:=d+e either after the assignment to a or after the assignment to b, it would be incorrect to replace c:=d+e by either c:=a or by c:=b.

The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy statement f:=g. For example, the assignment x:=t3 in block B5 of Fig. is a copy. Copy propagation applied to B5 yields:

x:=t3
a[t2]:=t5
a[t4]:=t3
goto B2



Copies introduced during common subexpression elimination.

This may not appear to be an improvement, but as we shall see, it gives us the opportunity to eliminate the assignment to x.

Dead-Code Eliminations

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. For example, we

discussed the use of debug that is set to true or false at various points in the program, and used in statements like

If (debug) print …

By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement, the value of debug is false. Usually, it is because there is one particular statement

Debug :=false

That we can deduce to be the last assignment to debug prior to the test no matter what sequence of branches the program actually takes. If copy propagation replaces debug by false, then the print statement is dead because it cannot be reached. We can eliminate both the test and printing from the o9bject code. More generally, deducing at compile time that the value of an expression is a co9nstant and using the constant instead is known as constant folding.

One advantage of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to x and transforms 1.1 into

a [t2] := t5
a [t4] := t3
goto B2

Loop Optimizations

We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop. Three techniques are important for loop optimization: code motion, which moves code outside a loop; induction-variable elimination, which we apply to eliminate I and j from the inner loops B2 and B3 and, reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

While (i<= limit-2)

Code motion will result in the equivalent of

t= limit-2;
while (i<=t)

Induction Variables and Reduction in Strength

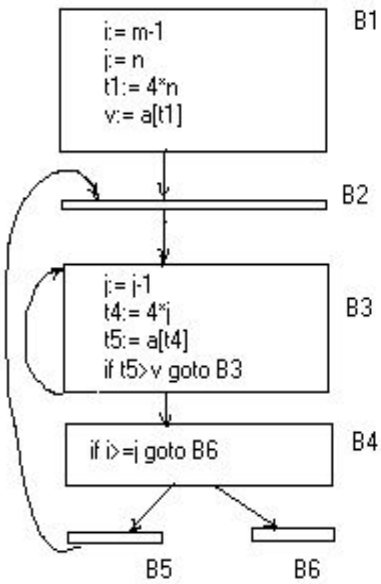
While code motion is not applicable to the quicksort example we have been considering the other two transformations are.Loops are usually processed inside out.For example consider the loop around B3.

Note that the values of j and t4 remain in lock-step;every time the value of j decreases by 1 ,that of t4 decreases by 4 because 4*j is assigned to t4.Such identifiers are called induction variables.

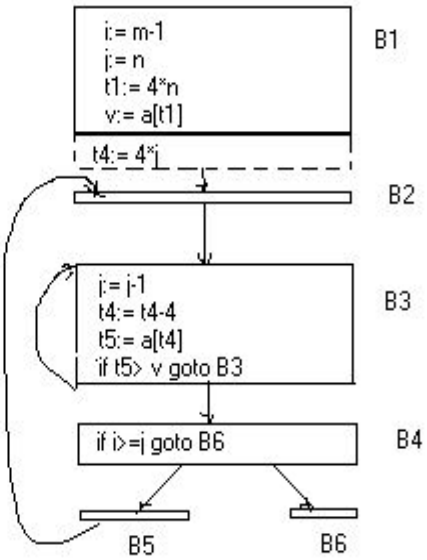
When there are two or more induction variables in a loop, iit may be possible to get rid of all but one, by the process of induction-variable elimination.For the inner loop around B3 in Fig. we cannot ger rid of either j or t4 completely.; t4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

Example: As the relationship t4:=4*j surely holds after such an assignment to t4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an intializations\ of t4 at the end of the blcok where j itself is initialized, shown by the dashed addt\ition to block B1 in second Fig.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.



Before



After

strength reduction applied to 4*j in block B3