COMPILER DESIGN

LECTURE NOTES ON COMPILER DESIGN

Prepared by

Dr. Subasish Mohapatra

Welcome

Department of Computer Science and Application
College of Engineering and Technology, Bhubaneswar
Biju Patnaik University of Technology, Odisha

SYLLABUS

Compiler Design (3-0-0)

MODULE – 1 (Lecture hours: 13)

Introduction: Overview and phases of compilation. **(2-hours)**

Lexical Analysis: Non-deterministic and deterministic finite automata (NFA & DFA), regular grammar, regular expressions and regular languages, design of a lexical analyzer as a DFA, lexical analyser generator. **(3-hours)**

Syntax Analysis: Role of a parser, context free grammars and context free languages, parse trees and derivations, ambiguous grammar.

Top Down Parsing: Recursive descent parsing, LL(1) grammars, non-recursive predictive parsing, error reporting and recovery.

Bottom Up Parsing: Handle pruning and shift reduces parsing, SLR parsers and construction or SLR parsing tables, LR(1) parsers and construction of LR(1) parsing tables, LALR parsers and construction of efficient LALR parsing tables, parsing using ambiguous grammars, error reporting and recovery, parser generator. **(8-hours)**

MODULE – 2 (Lecture hours: 14)

Syntax Directed Translation: Syntax directed definitions (SDD), inherited and synthesized attributes, dependency graphs, evaluation orders for SDD, semantic rules, application of syntax directed translation. **(5-hours)**

Symbol Table: Structure and features of symbol tables, symbol attributes and scopes. **(2-hours)**

Intermediate Code Generation: DAG for expressions, three address codes - quadruples and triples, types and declarations, translation of expressions, array references, type checking and conversions, translation of Boolean expressions and control flow statements, back patching, intermediate code generation for procedures. **(7-hours)**

MODULE – 3 (Lecture hours: 8)

Run Time Environment: storage organizations, static and dynamic storage allocations, stack allocation, handlings of activation records for calling sequences. **(3-hours)**

Code Generations: Factors involved, registers allocation, simple code generation using stack allocation, basic blocks and flow graphs, simple code generation using flow graphs. **(3-hours)**

Elements of Code Optimization: Objective, peephole optimization, concepts of elimination of local common sub-expressions, redundant and un-reachable codes, basics of flow of control optimization.

(2-hours)

CONTENTS

Module-1

Lecture #1

INTRODUCTION TO COMPILERS AND ITS PHASES

A compiler is a program takes a program written in a source language and translates it into an equivalent program in a target language. The source language is a high level language and target language is machine language.

Source program -> COMPILER -> Target program

Necessity of compiler

- 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.

Properties of Compiler

a) Correctness
   i) Correct output in execution.
   ii) It should report errors
   iii) Correctly report if the programmer is not following language syntax.
b) Efficiency
c) Compile time and execution.
d) Debugging / Usability.

Compiler	**Interpreter**
1. It translates the whole program at a time.	1. It translate statement by statement.
2. Compiler is faster.	2. Interpreter is slower.
3. Debugging is not easy.	3. Debugging is easy.
4. Compilers are not portable.	4. Interpreter are portable.

Types of compiler

1) **Native code compiler**
   A compiler may produce binary output to run /execute on the same computer and operating system. This type of compiler is called as native code compiler.
2) **Cross Compiler**
   A cross compiler is a compiler that runs on one machine and produce object code for another machine.
3) **Bootstrap compiler**
   If a compiler has been implemented in its own language . self-hosting compiler.

4) **One pass compiler**

The compilation is done in one pass over the source program, hence the compilation is completed very quickly. This is used for the programming language PASCAL, COBOL, FORTAN.

5) **Multi-pass compiler(2 or 3 pass compiler)**

In this compiler , the compilation is done step by step . Each step uses the result of the previous step and it creates another intermediate result.

Example:- gcc , Turboo C++

6) **JIT Compiler**

This compiler is used for JAVA programming language and Microsoft .NET

7) **Source to source compiler**

It is a type of compiler that takes a high level language as a input and its output as high level language. Example Open MP

List of compiler

1. Ada compiler
2. ALGOL compiler
3. BASIC compiler
4. C# compiler
5. C compiler
6. C++ compiler
7. COBOL compiler
8. Smalltalk comiler
9. Java compiler

Lecture #2

OVERVIEW OF LANGUAGE PROCESSING SYSTEM

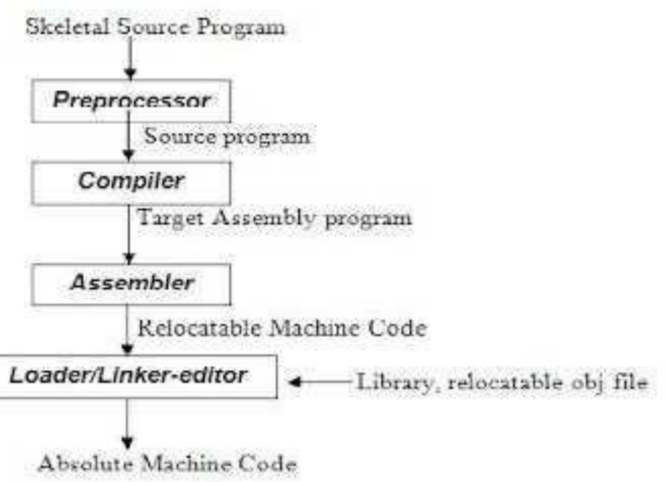


Fig 1.1 Language –processing System

A source program may be divided into modules stored in separate files.

Preprocessor –collects all the separate files to the source program.

A preprocessor produce input to compilers. They may perform the following functions.

1. *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text.
3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.

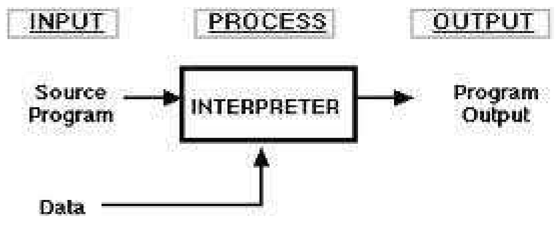
3. *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

ASSEMBLER

Programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

INTERPRETER

An interpreter is a program that appears to execute a source program as if it were machine language



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Synatx analysis
3. Semantic analysis
4. Direct Execution

Advantages

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes a various may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages

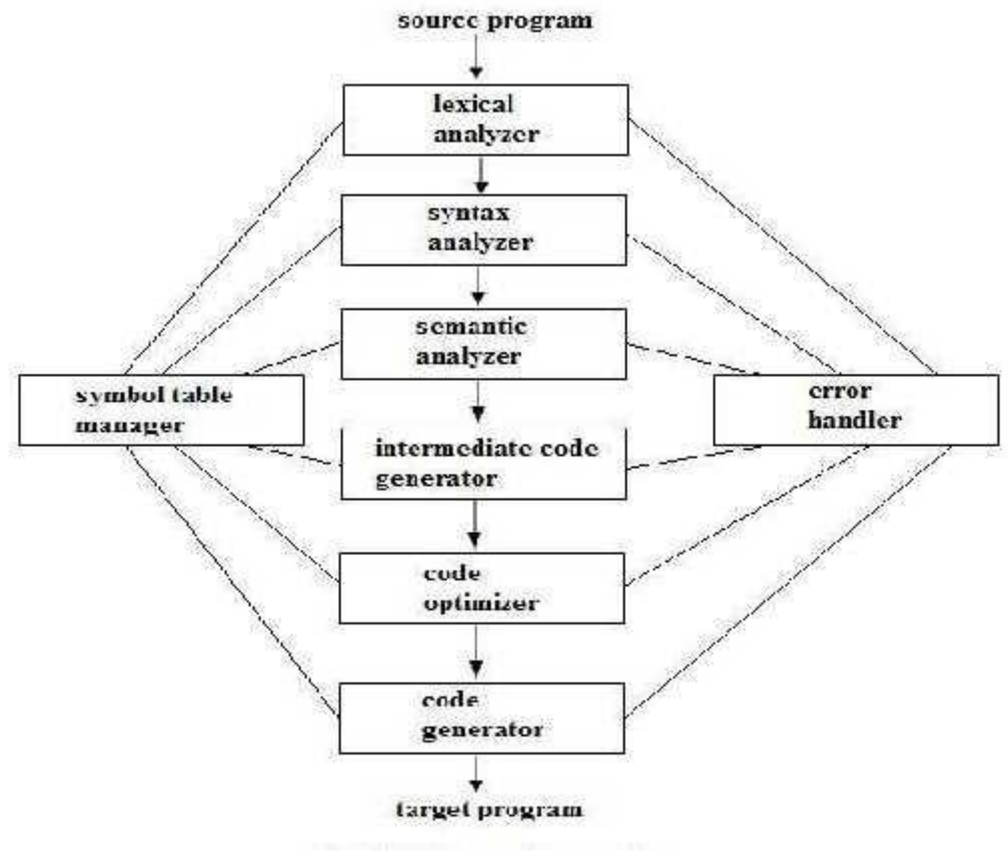
- The execution of the program is *slower*.
- Memory consumption is more.

Loader and Linker

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user's program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To over come this problems of wasted translation time and memory. System programmers developed another component called Loader

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could" relocate" directly behind the user's program. The task of adjusting programs o they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

STRUCTURE OF THE COMPILER DESIGN



Major Parts of a Compiler

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 phases in this part.
- In synthesis phase, the equivalent target program is created from this intermediate representation.

  Intermediate Code Generator, Code Generator, and Code Optimizer are the phases in this part.

$$\begin{tikzcd} \textit{Source Program} \arrow[r] & \text{Lexical Analyzer} \arrow[r] & \text{Syntax Analyzer} \arrow[r] & \text{Semantic Analyzer} \arrow[r] & \text{Intermediate Code Generator} \arrow[r] & \text{Code Optimizer} \arrow[r] & \text{Code Generator} \arrow[r] & \textit{Target Program} \end{tikzcd}$$

Lecture #3

Phases of a Compiler

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

Lexical Analyzer

- Lexical Analyzer reads the source program character by character and returns the *tokens* of the source program.

- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimiters and so on)

Example:

In the line of code *newval := oldval + 12*, tokens are:

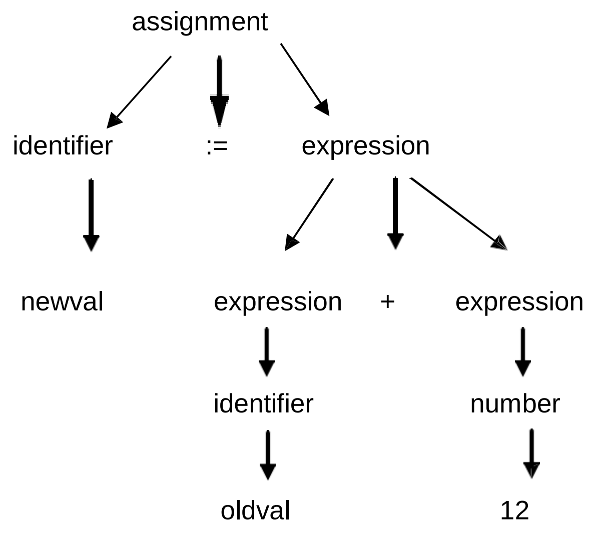
newval	(identifier)
:=	(assignment operator)
oldval	(identifier)
+	(add operator)
12	(a number)

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Syntax Analyzer

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

Example:
For the line of code *newval := oldval + 12*, parse tree will be:



- 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.

  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

Example:

CFG used for the above parse tree is:

assignment-> identifier := expression
expression -> identifier
expression -> number
expression -> expression + expression

- 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.
- 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 and Attribute grammars

Example:

In the line of code *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.
- These intermediate codes are generally machine architecture independent. But the level of intermediate codes is close to the level of machine codes.

Example:

*newval := oldval * fact + 1*

$\downarrow$

*id1 := id2 * id3 + 1*

$\downarrow$

```
MULT        id2, id3, temp1
ADDtemp1, #1, temp2
MOV         temp2, id1
```

The last form is the Intermediates Code (Quadruples)

Code Optimizer

The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

Example:
The above piece of intermediate code can be reduced as follows:

```
MULT id2, id3, temp1
ADD     temp1, #1, id1
```

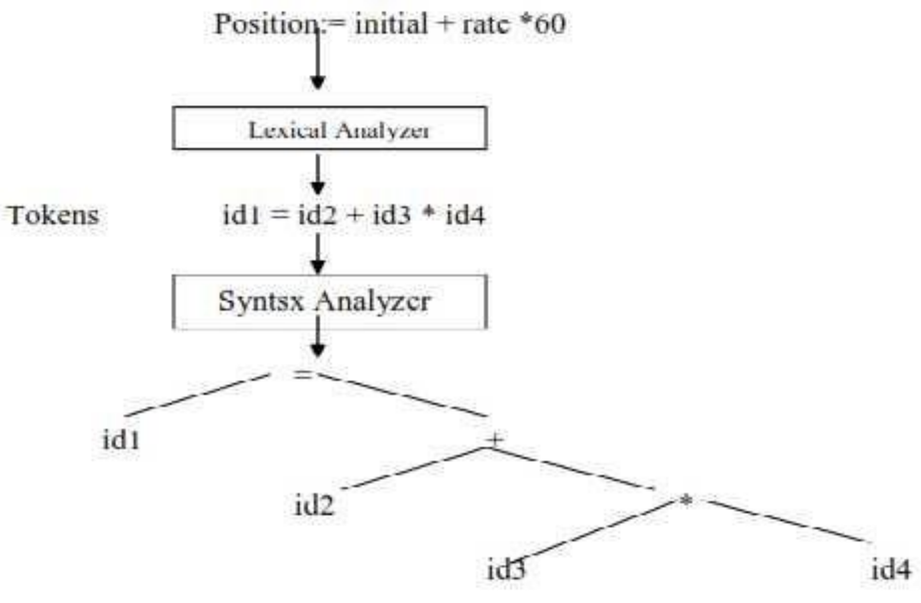
Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

Example:
Assuming that we have architecture with instructions that have at least one operand as a machine register, the Final Code our line of code will be:

```
MOVE        id2, R1
MULT        id3, R1
ADD          #1, R1
MOVE        R1, id1
```

Ex:-



Phases of a compiler are the sub-tasks that must be performed to complete the compilation process. Passes refer to the number of times the compiler has to traverse through the entire program.

Symbol Table Management:

A symbol table is a data structure that contains a record for each identifier with field for attributes of the identifier.

The type information about the identifier is detected during the lexical analysis phases and is entered into the symbol table.

Position= initial + rate*60;

Address	Symbol	Location	attributes
1	Position	1000	id, float
2	Intial	2000	id, float
3	Rate	3000	id, float
4	60	4000	constant, int

Error Detection and Reporting:

Each phase detects/encounters errors after detecting errors.

This phase must deal with errors to continue with the process of compilation.

The following are some errors encountered in each phase:

i) Lexical Analyzer- Miss spell token.

ii) Semantic Analyzer- Type Mismatch.

iii) Syntax Analyzer-Missing parenthesis , less no. of operands.

iv) Intermediate code generation – In compatible operands for an operand.

v) Code optimization- Unreachable statement.

vi) Code Generation- Memory restriction to store a variable.

Lecture #4

Languages

Terminology

- 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 all possible identifiers is a language.
- Operators on Strings:
  - *Concatenation*: xy represents the concatenation of strings x and y. $s\,\varepsilon = s$ $\varepsilon\, s = s$
  - $s^n = s\,s\,s \,..\, s$ (n times) $s^0 = \varepsilon$

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: $L^+ =$

Examples:

- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$
- $L_1L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$
- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$
- 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 and Finite Automata

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.

For Regular Expressions over alphabet Σ

Regular Expression	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)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$
- We may remove parentheses by using precedence rules.
  - * highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$

Examples:

– $\Sigma = \{0,1\}$

– $0|1 = \{0,1\}$

– $(0|1)(0|1) = \{00,01,10,11\}$

– $0^* = \{\varepsilon ,0,00,000,0000,....\}$

– $(0|1)^*$ = All strings with 0 and 1, including the empty string

Finite Automata

- A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic (DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.

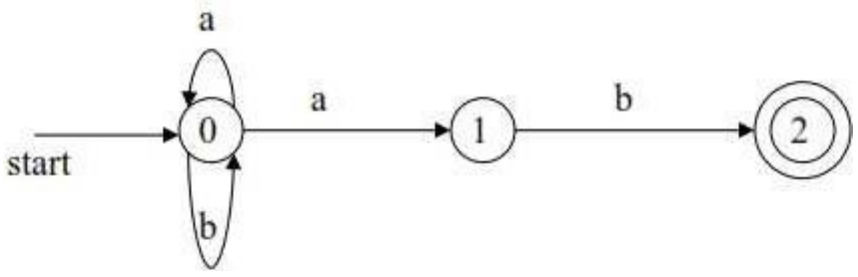
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - S - a set of states
  - Σ - a set of input symbols (alphabet)
  - move - a transition function move to map state-symbol pairs to sets of states.
  - s_0 - a start (initial) state
  - F- a set of accepting states (final states)
- ε- transitions are allowed in NFAs. In other words, we can move from one state to another

  one
- without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

Example:



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

Transition Function:

	a	b
0	{0,1}	{0}
1	{}	{2}
2	{}	{}

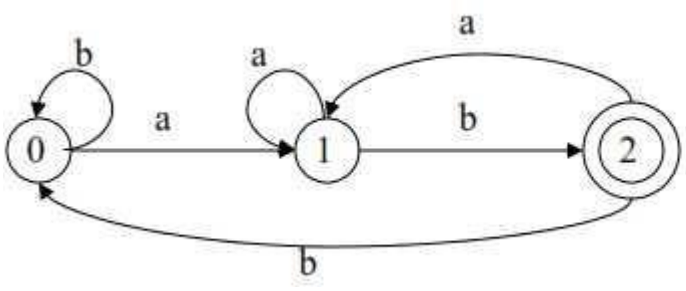
The language recognized by this NFA is (a|b)*ab

Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
- No state has ε- transition
- For each symbol a and state s, there is at most one labeled edge a leaving s. i.e. transition
- function is from pair of state-symbol to state (not set of states)

Example:

The DFA to recognize the language (a|b)* ab is as follows.



0 is the start state s0
{2} is the set of final states F

Σ = {a,b}

S = {0,1,2}

Transition Function:

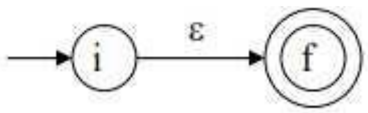
	a	B
0	1	0
1	1	2
2	1	0

Note that the entries in this function are single value and not set of values (unlike NFA).

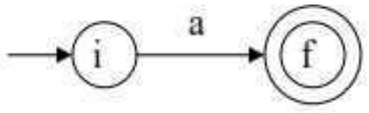
Lecture #5 \
Converting RE to NFA (Thomson Construction)

- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
- It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA.

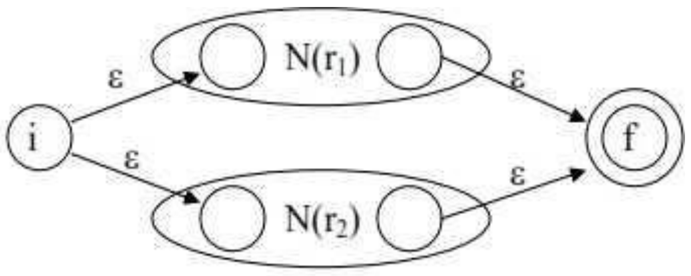
To recognize an empty string ε:



To recognize a symbol a in the alphabet Σ:

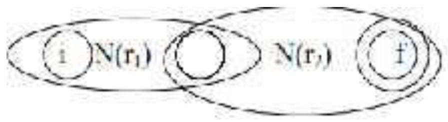


For regular expression r1 | r2:



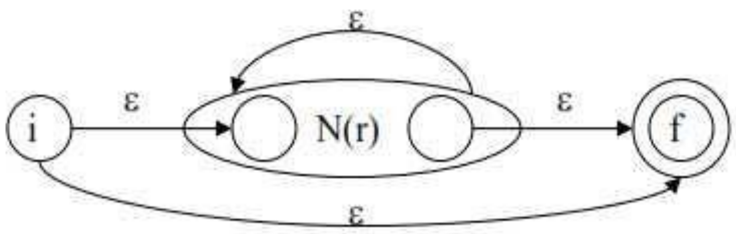
N(r1) and N(r2) are NFAs for regular expressions r1 and r2.

For regular expression r1. r2



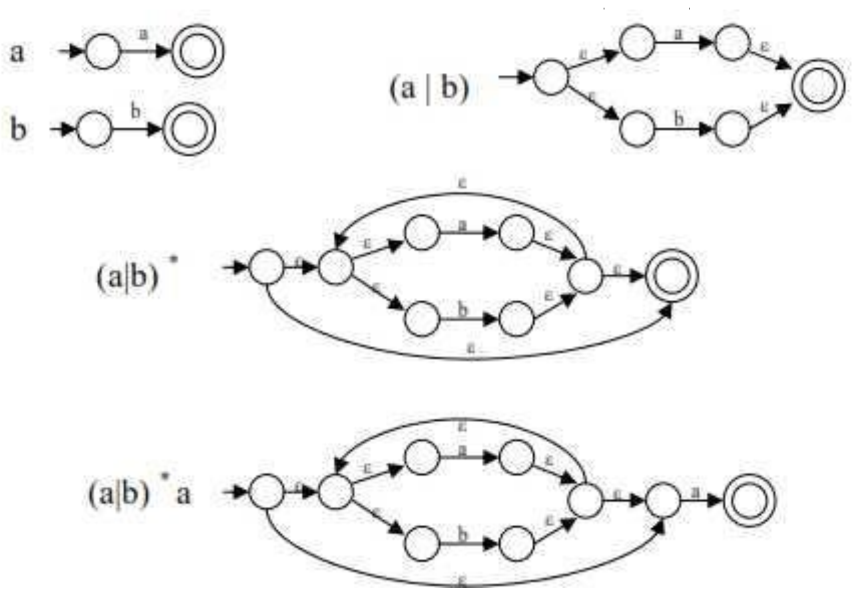
Here, final state of N(r1) becomes the final state of N(r1r2)

For regular expression r*



Example:

For a RE (a|b) * a, the NFA construction is shown below.



Converting NFA to DFA (Subset Construction)

We merge together NFA states by looking at them from the point of view of the input characters:

From the point of view of the input, any two states that are connected by an ε-transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an -transition will be represented by the same states in the DFA.

If it is possible to have multiple transitions based on the same symbol, then we can regard a transition on a symbol as moving from a state to a set of states (ie. the union of all those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.

To perform this operation, let us define two functions:

- The ε-**closure** function takes a state and returns the set of states reachable from it based on (one or more) -transitions. Note that this will always include the state tself.

We should be able to get from a state to any state in its ε-closure without consuming any input.

- The function **move** takes a state and a character, and returns the set of states reachable by one transition on this character.

We can generalize both these functions to apply to sets of states by taking the union of the application to individual states.

For Example, if A, B and C are states, move ({A,B,C},`a') = move(A, `a') U move(B, `a') U move (C,`a').

The Subset Construction Algorithm is a follows:

```
put ε-closure({s0}) as an unmarked state into the set of DFA (DS)

while (there is one unmarked S1 in DS) do

begin
        mark S1
        for each input symbol a do begin
        S2<-ε-closure(move(S1,a))
        if (S2 is not in DS) then add S2 into DS as an unmarked state transfunc[S1,a]<-S2
        end
End
```

ε-closure(move(S1,b)) = ε-closure({5}) = {1,2,4,5,6,7}

= S2 transfunc[S1,a]<-S1 transfunc[S1,b]<-S2

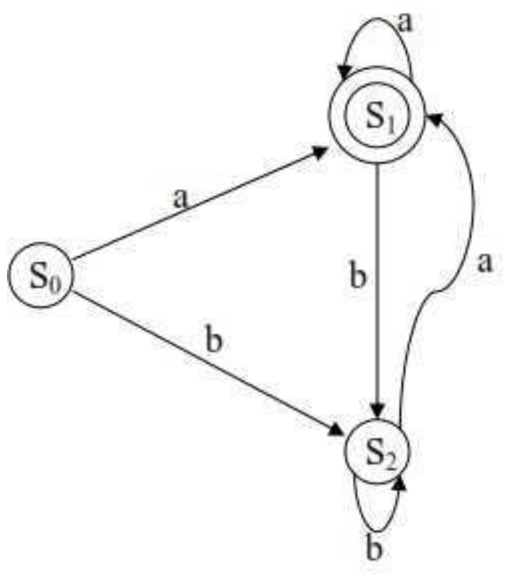
⇓mark S2

ε-closure(move(S2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S1

ε-closure(move(S2,b)) = ε-closure({5}) = {1,2,4,5,6,7}

= S2 transfunc[S2,a]<-S1 transfunc[S2,b]<-S2

S0 is the start state of DFA since 0 is a member of S0={0,1,2,4,7}
S1 is an accepting state of DFA since 8 is a member of S1 = {1,2,3,4,6,7,8}



Lecture #6
LEXICAL ANALYSIS

- Lexical Analyzer reads the source program character by character to produce tokens.
- Normally a lexical analyzer does not return a list of tokens at one shot; it returns a token when the parser asks a token from it.

Token, Pattern, Lexeme

- Token represents a set of strings described by a pattern. For example, an identifier represents a set of strings which start with a letter continues with letters and digits. The actual string is called as lexeme.

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

- For simplicity, a token may have a single attribute which holds the required information for that token. For identifiers, this attribute is a pointer to the symbol table, and the symbol table holds the actual attributes for that token.

- Examples:

– <identifier, attribute>	where attribute is pointer to the symbol table
– <assignment operator>	no attribute is needed
– <number, value>	where value is the actual value of the number

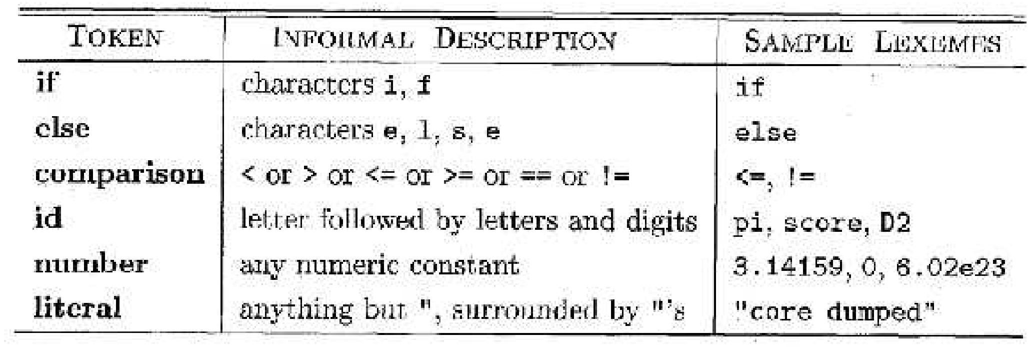
- Token type and its attribute uniquely identify a lexeme.
- *Regular expressions* are widely used to specify patterns.

Pattern:

A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

Lexeme:

A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.



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	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Lexical Analysis versus parsing

There are a number of reasons the analysis portion of a compiler is separated into lexical analysis and parsing (syntax analysis) phases.

- Simplicity of design. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer.

- Compiler efficiency is improved. specialized buffering techniques for reading input characters can speed up the compiler significantly.
- Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

Input Buffering(Lexical errors)

It is difficult to look one or more characters beyond the next lexeme before c o n f o r m the right lexeme.

There are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id.

In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large look aheads safely.

Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character. An important scheme involves two buffers that are alternately reloaded.

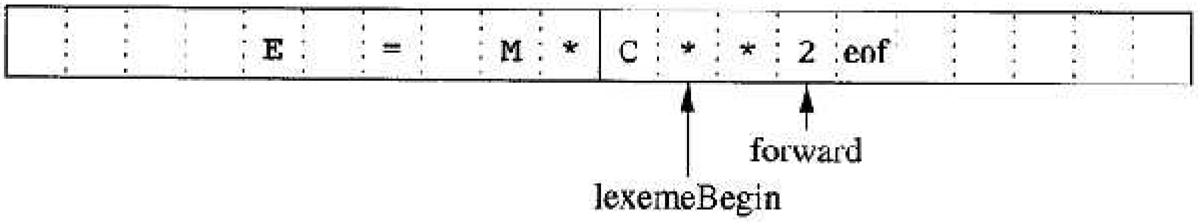


Fig: Using a Pair of Input Buffers

Two pointers to the input are maintained:

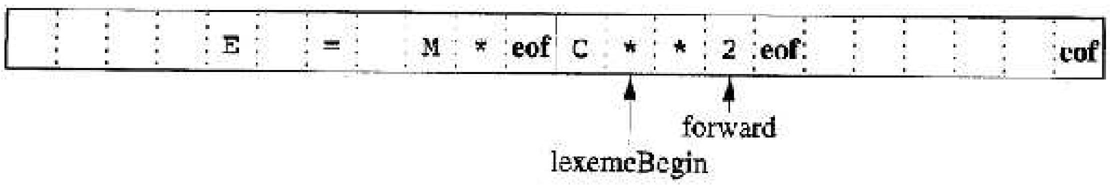
1. Pointer **lexemeBegin**, marks the beginning of the current lexeme
2. Pointer **forward** scans ahead until a pattern match is found.

Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig, forward has passed the end of the next lexeme, ** (the FORTRAN exponentiation operator), and must be retracted one position to its left.

Advancing forward requires that first test whether reached the end of one of the buffers, and if so, must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer.

Sentinels

for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read . We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof. Note that eof retains its use as a marker for the end of the entire input. Any eof that appears other than at the end of a buffer means that the input is at an end.



Panic Mode Error Recovery

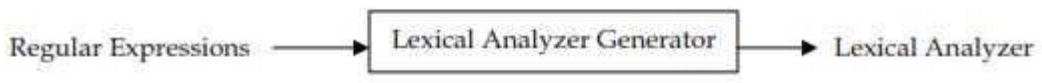
Suppose a situation arises in which a lexical analyzer is unable to proceed because non of the patterns for the token matches any prefix of the remaining in puts.

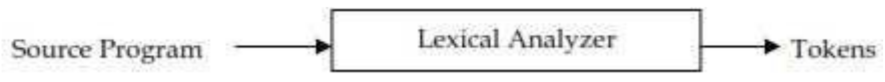
It detects successive characters from the remaining i/p until the lexical analyzer can find a well

defined token at the beginning of the i/p.
The other possible error recovery actions are:-

i) Delete 1 character from the remaining i/p.
ii) Insert a missing character to the remaining i/p.
iii) Replace a character by another character.
iv) Transpose two adjacent characters.

Lecture #7
Lexical Analyzer Generator





Tokens

LEX is an example of Lexical Analyzer Generator.

Input to LEX

- The input to LEX consists primarily of *Auxiliary Definitions* and *Translation Rules*.
- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use Auxiliary *Definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.
- An *Auxiliary Definition* is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

.

.

$d_n \rightarrow r_n$

where d_i is a distinct name and r_i is a regular expression over symbols in

$$\begin{tikzcd}
 & \Sigma \cup \{d_1,d_2,...,d_{i-1}\} \arrow[dl] \arrow[dr] & \\
\text{basic symbols} & & \text{previously defined names}
\end{tikzcd}$$

Example:

For 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)) *

Example:

For Unsigned numbers in Pascal digit → 0 | 1 | ... | 9 digits → digit $^+$

opt-fraction → (. digits) ?

opt-exponent → (E (+|-)? digits) ?

unsigned-num → digits opt-fraction opt-exponent

- *Translation Rules* comprise of a ordered list Regular Expressions and the Program Code to be executed in case of that Regular Expression encountered.

R_1	P_1
R_2	P_2
.	
.	
R_n	P_n

- The list is ordered i.e. the RE's should be checked in order. If a string matches more than one RE, the RE occurring higher in the list should be given preference and its Program Code is executed.

Implementation of LEX

•The Regular Expressions are converted into NFA's. The final states of each NFA correspond to some RE and its Program Code.

•Different NFA's are then converted to a single NFA with epsilon moves. Each final state of the NFA corresponds one-to-one to some final state of individual NFA's i.e. some RE and its Program Code. The final states have an order according to the corresponding RE's. If more than one final state is entered for some string, then the one that is higher in order is selected.

•This NFA is then converted to DFA. Each final state of DFA corresponds to a set of states (having at least one final state) of the NFA. The Program Code of each final state (of the DFA) is the program code corresponding to the final state that is highest in order out of all the final states in the set of states (of NFA) that make up this final state (of DFA).

Example:

AUXILIARY DEFINITIONS

(none)

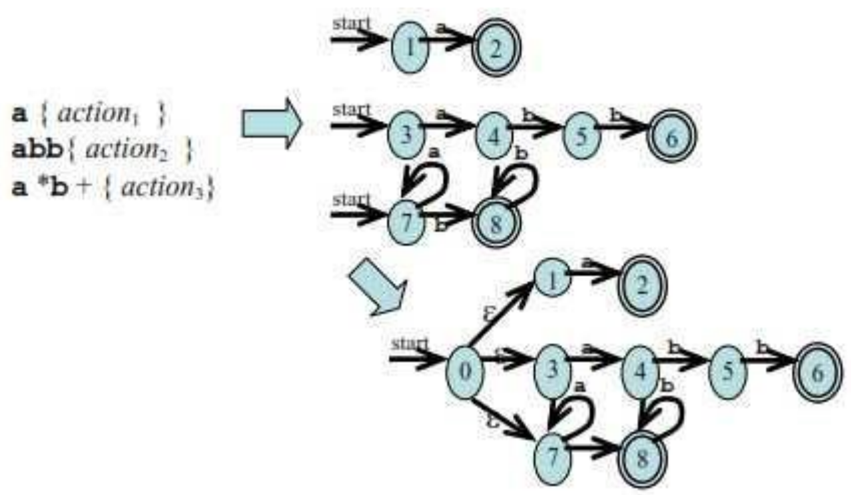
TRANSLATION RULES

a {Action_1}

abb{Action_2}

a^*b^+{Action_2}

First we construct an NFA for each RE and then convert this into a single NFA:



This NFA is now converted into a DFA. The transition table for the above DFA is as follows:

State	***A***	***b***	***Token found***
0137	247	8	None
247	7	58	a
8	-	8	a*b$^+$
7	7	8	None
58	-	68	a*b$^+$
68	-	8	abb

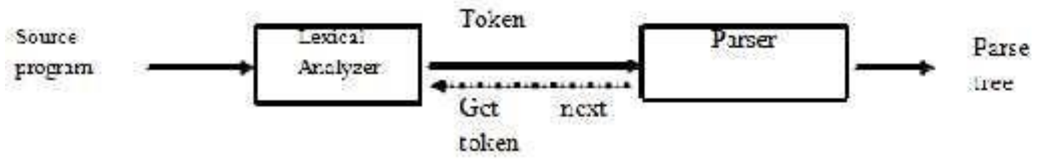
Lecture #8

BASICS OF SYNTAX ANALYSIS

- *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

- Parser works on a stream of tokens.
- The smallest item is a token.



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

Lecture #9
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

$A \to \alpha$ 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)

- 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
  (a) 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.

Derivations

Example:

(a) $E \to E + E \mid E – E \mid E * E \mid E / E \mid - E$

(b) $E \to (E)$

(c) $E \to id$

- $E \Rightarrow E+E$ means that E+E derives from E
  - we can replace E by E+E
  - to able to do this, we have to have a production rule $E \to E+E$ in our grammar.
- $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$ means that a sequence of replacements of non-terminal symbols
- In general a derivation step is
  $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if there is a production rule $A \to \gamma$ in our grammar

Where α and β are arbitrary strings of terminal and non-terminal Symbol

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

- 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.

Example:

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

- If we always choose the right-most non-terminal in each derivation step, this derivation is called as right-most derivation.

Example:

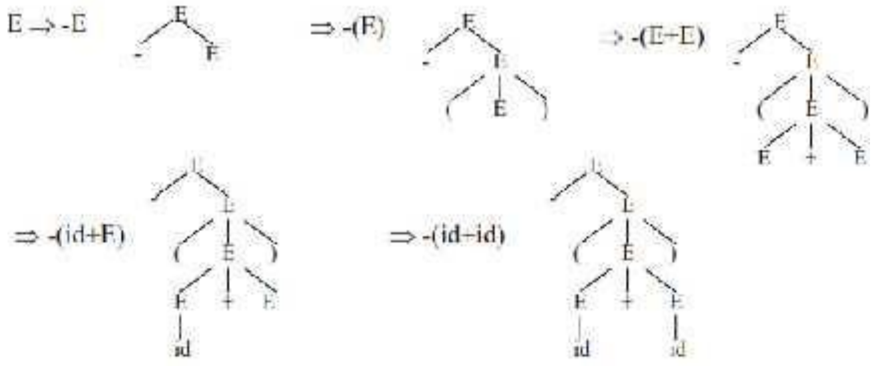
$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

- 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.

Example:



Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an *ambiguous*
- grammar.
- 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 unambiguous 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.
- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

Example:

To disambiguate the grammar

E → E+E | E*E | E^E | id | (E),

we can use precedence of operators as follows:

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

We get the following unambiguous grammar: E → E+T | T

T → T*F | F

F → G^F | G

G → id | (E)

Lecture #10

Left Recursion

- A grammar is *left recursive* if it has a non-terminal A such that there is a derivation: A ⇒ Aα 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 → A α | β where β does not start with A

⇓ Eliminate immediate left recursion

A → β A'

A' → α A' | ε an equivalent grammar

In general,

A → A α1 | ... | A αm | β1 | ... | βn where β1 ... βn do not start with A

⇓ Eliminate immediate left recursion

A → β1 A' | ... | βn A'

A' → α1 A' | ... | αm A' | ε an equivalent grammar

Example:

E → E+T | T T → T*F | F F → id | (E)

⇓ Eliminate immediate left recursion

E → T E'

E' → +T E' | ε

T → F T'

T' → *F T' | ε

F → id | (E)

•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.

Example:

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$

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.

Elimination

Arrange non-terminals in some order: A1 ... An

```
for i from 1 to n do {
for j from 1 to i-1 do  {
        replace each production
            $A_i \rightarrow A_j \gamma$
            by
                $A_i \rightarrow \alpha_1 \gamma \mid ... \mid \alpha_k \gamma$
                    where $A_j \rightarrow \alpha_1 \mid ... \mid \alpha_k$
        }
        eliminate immediate left-recursions among Ai productions
}
```

Example:

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid f$

Case 1: Order of non-terminals: S, A

for S:
- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:
- Replace A → Sd with A → Aad | bd

So, we will have A → Ac | Aad | bd | f

- Eliminate the immediate left-recursion in

A A → bdA’ | fA’

A’ → cA’ | adA’ | ε

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

S → Aa | b

A → bdA’ | fA’

A’ → cA’ | adA’ | ε

Case 2: Order of non-terminals: A, S

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

A → SdA’ | fA’

A’ → cA’ | ε

for S:
- Replace S → Aa with S → SdA’a | fA’a

So, we will have S → SdA’a | fA’a | b

- Eliminate the immediate left-recursion in S

S → fA’aS’ | bS’

S’ → dA’aS’ | ε

So, the resulting equivalent grammar which is not left-recursive is: S → fA’aS’ | bSS’ → dA’aS’ | ε

A → SdA’ | fA’

A’ → cA’ | ε

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 → βα1 | βα2

where α is non-empty and the first symbols of β1 and β2 (if they have one)are different.

• when processing α we cannot know whether expand

A to βα1 or

A to βα2

• But, if we re-write the grammar as follows

A → αA’

A’ → β1 | β2 so, we can immediately expand A to αA’

10.1 Algorithm

• For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

A → βα1 | ... | βαn | γ1 | ... | γm convert it into

A → αA’ | γ1 | ... | γm

A’ → β1 | ... | βn

Example:

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Example:

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

Lecture #11
YACC

YACC generates C code for a syntax analyzer, or parser. YACC uses grammar rules that allow it to analyze tokens from LEX and create a syntax tree. A syntax tree imposes a hierarchical structure on tokens. For example, operator precedence and associativity are apparent in the syntax tree. The next step, code generation, does a depth-first walk of the syntax tree to generate code. Some compilers produce machine code, while others output assembly.

YACC takes a default action when there is a conflict. For shift-reduce conflicts, YACC will shift. For reduce-reduce conflicts, it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous.

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Input to YACC is divided into three sections. The definitions section consists of token declarations, and C code bracketed by “%{“ and “%}”. The BNF grammar is placed in the rules section, and user subroutines are added in the subroutines section.

Lecture #12

TOP-DOWN PARSING

- 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

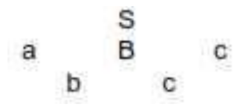
• No backtracking
• Efficient
• Needs a special form of grammars i.e. LL (1) grammars.
• Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
• Non-Recursive (Table Driven) Predictive Parser is also known as LL (1) parser.

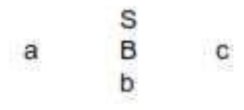
Recursive-Descent Parsing (uses Backtracking)

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

Example:

If the grammar is S → aBc; B → bc | b and the input is abc:





Predictive Parser

Grammar	--------------->	------------->	a grammar suitable for predictive
	Eliminate	Left	parsing (a LL(1) grammar)
	left recursion	Factor	no %100 guarantee.

- 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.

Example:

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

When we are trying to write the non-terminal *stmt*, 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).

Lecture #13

Recursive Predictive Parsing

Each non-terminal corresponds to a procedure.

Example:

A → aBb | bAB

```
proc A {
        case of the current token {
        'a': - match the current token with a, and move to the next token;
        - call 'B';
        - match the current token with b, and move to the next token;
        'b': - match the current token with b, and move to the next token;
        - call 'A';
        - call 'B';
}
}
```

13.1 Applying ε-productions

A → aA | bB | ε

If all other productions fail, we should apply an ε-production. For example, if the current token is not a or b, we may apply the ε-production.

Most correct choice: We should apply an ε-production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).

Example:

A → aBe | cBd | C

B → bB | ε

C → f

```
proc A {
        case of the current token {

        a:      - match the current token with a and move to the next token;
                - call B;
                - match the current token with e and move to the next token;
        c:      - match the current token with c and move to the next token;
                - call B;
                - match the current token with d and move to the next token;
        f:      - call C                        //First Set of C
        }
        }
proc C {

}
        match the current token with f and move to the next token;
proc B {



        case of the current token {
        b:      - match the current token with b and move to the next token;
        - call B
        e,d:    - do nothing                    //Follow Set of B

        }

        }
```

Lecture #14

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.

$$\begin{tikzcd}
 & \text{input buffer} \arrow[d, shift left] & \\
\text{stack} \arrow[r] & \text{Non-recursive Predictive Parser} \arrow[u, shift left] \arrow[r] \arrow[d, shift left] & \text{output} \\
 & \text{Parsing Table} \arrow[u, shift left] &
\end{tikzcd}$$

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 (i.e. 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.

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.

- If X and a are $- > parser halts (successful completion)

- 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.

- If X is a non-terminal

- parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule

  $X \rightarrow Y_1 Y_2 \ldots Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, \ldots, Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$ to represent a step of the derivation.

- 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.

Example:

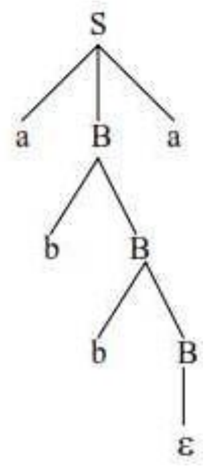
For the Grammar is S → aBa; B → bB | ε and the following LL(1) parsing table:

	A	**B**	**$**
S	S → aBa		
B	B → ε	B → bB	

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

Outputs: S → aBa   B → bB   B → bB   B → ε

Derivation (left-most): S⇒aBa⇒abBa⇒abbBa⇒abba



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 ⇒ αAaβ
  - $ is in FOLLOW(A) if S ⇒ αA

To Compute FIRST for Any String X:

- If X is a terminal symbol->FIRST(X)={X}
- If X is a non-terminal symbol and X → ε is a production rule->ε is in FIRST(X).
- If X is a non-terminal symbol and X → $Y_1Y_2..Y_n$ is a production rule

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

To Compute FOLLOW (for non-terminals):

- If S is the start symbol - > $ is in FOLLOW(S)
- If A → αBβ is a production rule->everything in FIRST(β) is FOLLOW(B) except ε
- If (A → αB is a production rule) or (A → αBβ is a production rule and ε is in FIRST(β))

  ->everything in FOLLOW(A) is in FOLLOW(B).
- Apply these rules until nothing more can be added to any follow set.

Algorithm for Constructing LL(1) Parsing Table:

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

Example:

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

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

LL(1) Parsing Table

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

	id	+	*	**(**	**)**	**$**
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Lecture #15

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.
- 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 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).

Non- LL(1) Grammars

Example:

$S \rightarrow i\ C\ t\ S\ E \mid a$

$E \rightarrow e\ S \mid \varepsilon$

$C \rightarrow b$

FOLLOW(S) = { \$,e }
FOLLOW(E) = { \$,e }
FOLLOW(C) = { t }

FIRST(iCtSE) = {i}
FIRST(a) = {a}
FIRST(eS) = {e}
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]

The Problem with multiple entries here is that of 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 its (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 → αβ1 | αβ2
  - ->any terminal that appears in FIRST(αβ1) also appears in FIRST(αβ2).
- An ambiguous grammar cannot be a LL(1) grammar.

Lecture #16

BASIC BOTTOM-UP PARSING TECHNIQUES

- 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.

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

  (b) $\leftarrow$ (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.
- 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.

Example:
For Grammar $S \rightarrow aABb$; $A \rightarrow aA \mid a$; $B \rightarrow bB \mid b$ and Input string aaabb,

$$\begin{aligned} & aaabb \\ \Rightarrow\ & aaAbb \\ \Rightarrow\ & aAbb \\ \Rightarrow\ & aABb \\ \Rightarrow\ & S \end{aligned}$$

The above reduction corresponds to the following rightmost derivation: $S \Rightarrow aABb \Rightarrow aAbb \Rightarrow aaAbb \Rightarrow aaabb$

Handle

- Informally, a handle of a string is a substring that matches the right side of a production rule.
  - But not every substring that 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

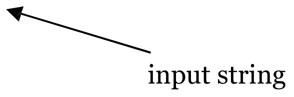
rightmost derivation of γ.

$$S \Rightarrow \alpha A \omega \Rightarrow \alpha\beta\omega$$

- 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.

- A right-most derivation in reverse can be obtained by handle-pruning.

$S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \omega$



- 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.

Example:

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

Right-Most Derivation of id+id*id is

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

```
Right-Most  Sentential  Form                    Reducing  Production  id+id*id
             F → id
F+id*id                                                        T  →  F  T+id*id
E  → T  E+id*id                                                 F  →  id E+F*id
T  → F  E+T*id                                                   F  →  id E+T*F
T → T*F E+T                                     E → E+T E
```

Handles are underlined in the right-sentential forms.

Stack Implementation

- There are four possible actions of a shift-parser action:
- Shift : The next input symbol is shifted onto the top of the stack.
- Reduce: Replace the handle on the top of the stack by the non-terminal.
- Accept: Successful completion of parsing.
- 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 $.

Example:

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

Lecture #17

<u>Conflicts during Shift Reduce Parsing</u>

- 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.

$$\begin{tikzcd}
 & \text{LR (k)} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{input scanned from left to right} & \text{Right most derivation} & \boxed{\text{k input symbols used as a look- head symbol do determine parser action}}
\end{tikzcd}$$

- An ambiguous grammar can never be a LR grammar.

Types of Shift Reduce Parsing

There are two main categories of shift-reduce parsers

1. Operator-Precedence Parser

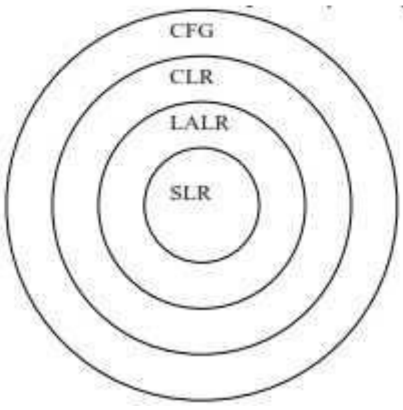
– simple, but only a small class of grammars.

2. LR-Parsers

– Covers wide range of grammars.

- SLR – Simple LR parser
- CLR – most general LR parser (Canonical LR)
- LALR – intermediate LR parser (Look Ahead LR)

– SLR, CLR and LALR work same, only their parsing tables are different.



Lecture #18

Operator Precedence Parsing

- Operator grammar
– small, but an important class of grammars
– we may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.
- In an *operator grammar*, no production rule can have:
– ε at the right side
– two adjacent non-terminals at the right side.

Examples:

E→AB	E→EOE	E→E+E |
A→a	E→id	E*E |
B→b	O→+|*|/	E/E | id
not operator grammar	not operator grammar	operator grammar

Precedence Relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

a <· b b has higher precedence than a a =· b b has same precedence as a
a ·> b b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators. (Unary minus causes a problem).
- The intention of the precedence relations is to find the handle of a right-sentential form,

  <· with marking the left end,
  =· appearing in the interior of the handle, and
  ·> marking the right hand.
- In our input string **$**$a_1a_2...a_n$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Example:

E → E+E | E-E | E*E | E/E | E^E | (E) | -E | id

The partial operator-precedence table for this grammar is as shown.

	id	+	*	$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Then the input string id+id*id with the precedence relations inserted will be:

$ <. id .> + <. id .> * <. id .> $

Using Precedence relations to find Handles

- Scan the string from left end until the first ·> is encountered.
- Then scan backwards (to the left) over any =· until a <· is encountered.
- The handle contains everything to left of the first ·> and to the right of the <· is encountered.

The handles thus obtained can be used to shift reduce a given string.

Operator-Precedence Parsing Algorithm

- The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals

Parsing Algorithm

The input string is w$, the initial stack is $ and a table holds precedence relations between certain terminals.

```
set p to point to the first symbol of w$ ;
repeat forever
if ( $ is on top of the stack and p points to $ ) then return else {
let a be the topmost terminal symbol on the stack and let b be the symbol pointed to by p;
if ( a <. b or a =· b )then {        /* SHIFT */
push b onto the stack;
advance p to the next input symbol;
}
else if ( a .> b ) then                /* REDUCE */
repeat pop stack
until ( the top of stack terminal is related by <. to the terminal most recently popped);
else error();
}
```

Example:

stack	*input*	*action*
$	id+id*id$	$ <· id shift
$id	+id*id$	id ·> + reduce E → id
$	+id*id$	shift
$+	id*id$	shift
$+id	*id$	id ·> * reduce E → id
$+	*id$	shift
$+*	id$	shift
$+*id	$	id ·> $ reduce E → id
$+*	$	* ·> $ reduce E → E*E

$+	$	+ ·> $ reduce E → E+E
$	$	accept

Creating Operator-Precedence Relations from Associativity and Precedence

1. If operator O1 has higher precedence than operator O2,
->O1 .> O2 and O2 <. O1

2. If operator O1 and operator O2 have equal precedence, they are left-associative ->O1 .> O2 and O2 .> O1 they are right-associative- > O1 <. O2 and O2 <. O1

3. For all operators O,
O <. id, id .> O, O <. (, (<. O, O .>),) .> O, O .> $, and $ <. O

4. Also, let

(=·)	$ <. (	id .>)	) .> $
(<. (	$ <. id	id .> $	) .>)
(<. id			

Example:

The complete table for the Grammar E → E+E | E-E | E*E | E/E | E^E | (E) | -E | id is:

	+	-	*	/	^	id	(	)	$
+	.>	.>	<.	<.	<.	<.	<.	.>	.>
-	.>	.>	<.	<.	<.	<.	<.	.>	.>
*	.>	.>	.>	.>	<.	<.	<.	.>	.>
/	.>	.>	.>	.>	<.	<.	<.	.>	.>
^	.>	.>	.>	.>	<.	<.	<.	.>	.>
id	.>	.>	.>	.>	.>			.>	.>
(	<.	<.	<.	<.	<.	<.	<.	=·	
)	.>	.>	.>	.>	.>			.>	.>
$	<.	<.	<.	<.	<.	<.	<.		

Operator-Precedence Grammars

There is another more general way to compute precedence relations among terminals:

1. a = b if there is a right side of a production of the form αaβbγ, where β is either a single non-terminal or ε.
2. a < b if for some non-terminal A there is a right side of the form αaAβ and A derives to γbδ

where γ is a single non-terminal or ε.

3. a > b if for some non-terminal A there is a right side of the form αAbβ and A derives to γaδ where δ is a single non-terminal or ε.

Note that the grammar must be unambiguous for this method. Unlike the previous method, it does not take into account any other property and is based purely on grammar productions. An ambiguous grammar will result in multiple entries in the table and thus cannot be used.

Handling Unary Minus

- Operator-Precedence parsing cannot handle the unary minus when we also use the binary minus in our grammar.
- The best approach to solve this problem is to let the lexical analyzer handle this problem.
  - –The lexical analyzer will return two different operators for the unary minus and the binary minus.
  - –The lexical analyzer will need a look ahead to distinguish the binary minus from the unary minus.
- Then, we make

O <. unary-minus	for any operator
unary-minus .> O	if unary-minus has higher precedence than O
unary-minus <. O	if unary-minus has lower (or equal)
precedence than O	

Precedence Functions

- Compilers using operator precedence parsers do not need to store the table of precedence relations.
- The table can be encoded by two precedence functions f and g that map terminal symbols to integers.
- For symbols a and b.

  f(a) < g(b) whenever a <· b

  f(a) = g(b) whenever a =· b

  f(a) > g(b) whenever a ·> b

Advantages and Disadvantages

- Advantages:
  - simple
  - powerful enough for expressions in programming languages

- Disadvantages:
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

Lecture #19

LR PARSING

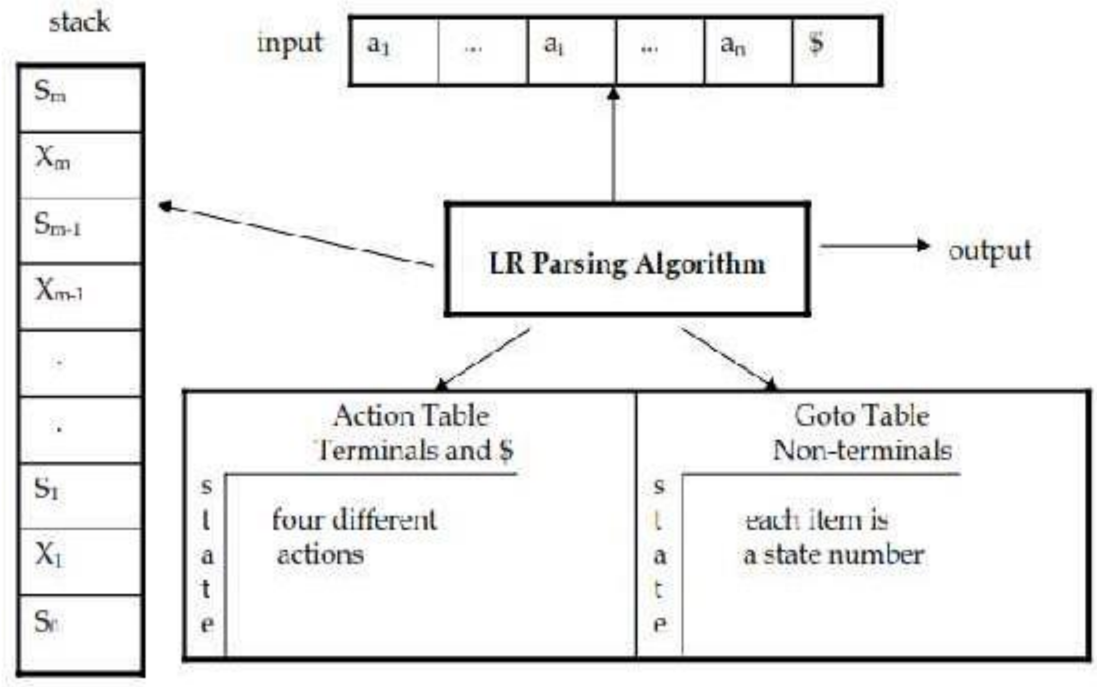
LR parsing is attractive because:

- 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.

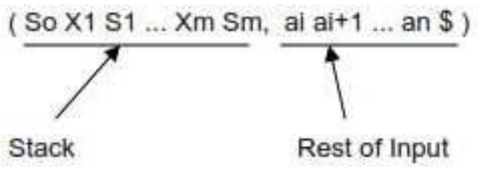
LL(1)-Grammars ⊂ LR(1)-Grammars

- 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.

Parser Configuration



• A configuration of a LR parsing is:



• Sm and ai decides the parser action by consulting the parsing action table. (*Initial Stack* contains just So)

• A configuration of a LR parsing represents the right sentential form:
X1 ... Xm ai ai+1 ... an \$

Parser Actions

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack
(So X1 S1 ... Xm Sm, ai ai+1 ... an $) ->(So X1 S1 ... Xm Sm ai s, ai+1 ... an $)

2. **reduce A→β** (or **rn** where n is a production number)

– pop 2|β| (=r) items from the stack; let us assume that β = Y1Y2...Yr

  – then push **A** and **s** where **s=goto[sm-r,A]**

(So X1 S1 ... Xm Sm, ai ai+1 ... an $)->(So X1 S1 ... Xm-r Sm-r A s, ai ... an $)

  – Output is the reducing production reduce A→β

  – In fact, Y1Y2...Yr is a handle.

X1 ... Xm-r A ai ... an $ ⇒ X1 ... Xm Y1...Yr ai ai+1 ... an $

3. **Accept** – Parsing successfully completed.
4. **Error** -- Parser detected an error (an empty entry in the action table) Example:
Let following be the grammar and its LR parsing table.

1) E → E+T

2) E → T

3) T → T*F

4) T → F

5) F → (E)

6) F → id

	Action							*Goto*		
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				

The action of the parser would be as follows:

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

Lecture #20

Constructing SLR Parsing tables

- An LR parser using SLR parsing tables for a grammar G is called as the SLR parser for G.
- If a grammar G has an SLR parsing table, it is called SLR grammar.
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.
- *Augmented Grammar*: G' is G with a new production rule S'→S where S' is the new starting symbol.

LR(0) Items

- An **LR(0) item** of a grammar G is a production of G a dot at the some position of the right side.

Example:

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.

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 → α.Bβ is in closure(I) and Bγ→ is a production rule of G; then B→.γ will be in the closure(I). We will apply this rule until no more new LR(0) items can be added to closure(I).

Example:

E' → E ; E → E+T;

E → T;

T → T*F; T → F;

F → (E);

F → id

closure({E' → .E}) = { E' → .E, E → .E+T, E → .T, T → .T*F, T → .F, F → .(E), F → .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 →** α**X.**β**})** will be in goto(I,X).

Example:

I = { E' → .E, E → .E+T, E → .T, T → .T*F, T → .F, F → .(E), F → .id }

goto(I,E) = { E' → E., E → E.+T }

goto(I,T) = { E → T., T → T.*F }

goto(I,F) = {T → F. }

goto(I,() = {F→ (.E), E→ .E+T, E→ .T, T→ .T*F, T→ .F, F→ .(E), F→ .id }

goto(I,id) = { F → id. }

Lecture #21

<u>Construction of The Canonical LR(0) Collection</u>

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'→.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. Example:
For grammar used above, Canonical LR(0) items are as follows-

I0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I1: E' → E.
E → E.+T

I2: E → T.
T → T.*F

I3: T → F.

I4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I7: T → T*.F
F → .(E)
F → .id

I8: F → (E.)
E → E.+T

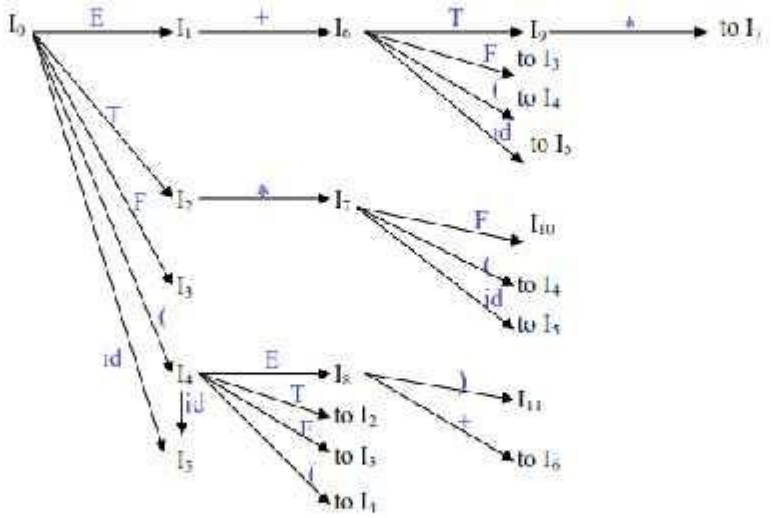
I9: E → E+T.
T → T.*F

I10: T → T*F.

I11: F → (E).

I5: F → id.

Transition Diagram (DFA) of GOTO Function is as follows-



Parsing Table

1. Construct the canonical collection of sets of LR(0) items for G'.

   C←{I0,...,In}

2. Create the parsing action table as follows

a. If a is a terminal, Aα→.aβ in Ii and goto(Ii,a)=Ij then action[i,a] is ***shift j*.**

b. If Aα→. is in Ii , then action[i,a] is ***reduce A***α→ for all a in FOLLOW(A) where

   A≠S'.

c. If S'→S. is in Ii , then action[i,$] is ***accept***.

d. If any conflicting actions generated by these rules, the grammar is not SLR(1).

3. Create the parsing goto table

a. for all non-terminals A, if goto(Ii,A)=Ij then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains S'→.S

Example:

For the Grammar used above, SLR Parsing table is as follows:

	Action							*Goto*		
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				

Lecture #22

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**.

Example:

S → L=R S → R	I_0: S' → .S S → .L=R	I_1: S' → S.	I_6: S → L=.R R → .L	I_9: S → L=R.
L→ *R L → id	S → .R L → .*R	I_2: S → L.=R R → L.	L→ .*R L → .id	
R → L	L → .id			
	R → .L	I_3: S → R.		
Problem in I_2		I_4: L → *.R R → .L	I_7: L → *R.	
FOLLOW(R)={=,$} = shift 6		L→ .*R L → .id	I_8: R → L.	
& reduce by R → L				
shift/reduce conflict		I_5: L → id.		

- 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**.

Example:

S → AaAb	I_0: S' → .S
S → BbBa	S → .AaAb
A → ε	S → .BbBa
B → ε	A → .
B → .	

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

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

If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Constructing Canonical LR(1) Parsing tables

- In SLR method, the state i makes a reduction by Aα→ when the current token is a:

– if the Aα→**.** in the Ii and a is FOLLOW(A)

- In some situations, βA cannot be followed by the terminal a in a right-sentential form when

αβ 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: item

  $A \rightarrow \alpha.\beta,a$ where a is the look-head of the LR(1) (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 Aα→ only if the next input symbol

is a (not for any terminal in FOLLOW(A)).

- A state will contain $A \rightarrow \alpha.,a1$ where $\{a1,...,an\} \subseteq$ FOLLOW(A)

...

$A \rightarrow \alpha.,an$

Closure and GOTO Operations

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

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

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

for each terminal b in FIRST(βa) .

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($\{A \rightarrow \alpha X.\beta,a\}$) will be in goto(I,X).

Lecture #23

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 set of LR(1) items containing the following items

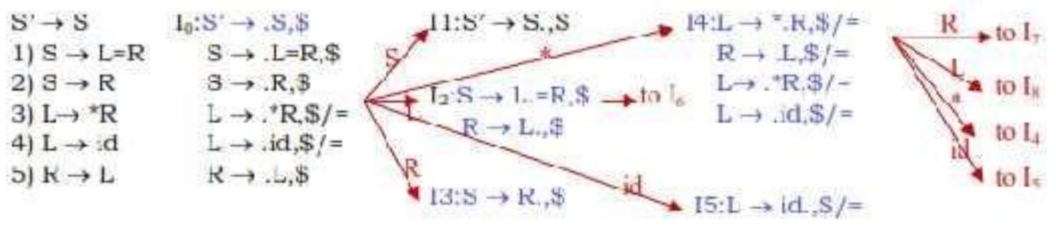
A → α.β,a1

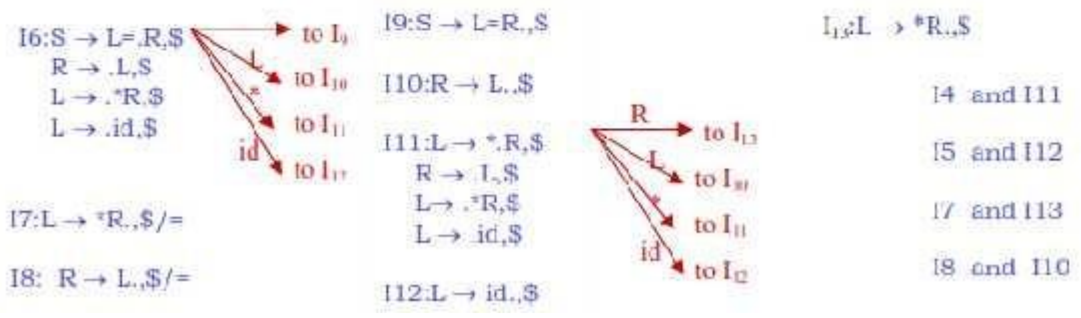
...

A → α.β,an

can be written as A → α.β,a1/a2/.../an

Example:





Parsing Table

1. Construct the canonical collection of sets of LR(1) items for G'.

C←{I0,...,In}

2. Create the parsing action table as follows

a. If a is a terminal, Aα→.aβ,b in Ii and goto(Ii,a)=Ij then action[i,a] is *shift j*. b. If Aα→.,a is in Ii , then action[i,a] is *reduce A*α→ where A≠S'.

c. If S'→S.,$ is in Ii , then action[i,$] is *accept*.

d. If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table

a. for all non-terminals A, if goto(Ii,A)=Ij then goto[i,A]=j

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains S'→.S,$

Example:

For the above used Grammar, the parse table is as follows:

	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			

Lecture #24

Constructing LALR Parsing tables

- **LALR** stands for **LookAhead LR.**
- LALR parsers are often used in practice because LALR parsing tables are smaller than Canonical LR 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.

Canonical LR(1) Parser -> LALR Parser shrink # of states

- This shrink process may introduce a **reduce/reduce** conflict in the resulting LALR parser. In that case the grammar is NOT LALR.
- This shrink process cannot 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.

Example:

$S \rightarrow L.=R,\$$ -> $S \rightarrow L.=R$ $R \rightarrow L.,\$$ $R \rightarrow L.$

- 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.

Example:

I_1:L → id.,= I_{12}: L → id.,=

L → 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.

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={I0,...,In} ->C'={J1,...,Jm} 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 $J=I1 \cup ... \cup Ik$ since I1,...,Ik have same cores

->cores of goto(I1,X),...,goto(I2,X) must be same.

–So, goto(J,X)=K where K is the union of all sets of items having same cores as goto(I1,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.

I1 : $A \rightarrow \alpha.,a$ I2: $A \rightarrow \alpha.,b$

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

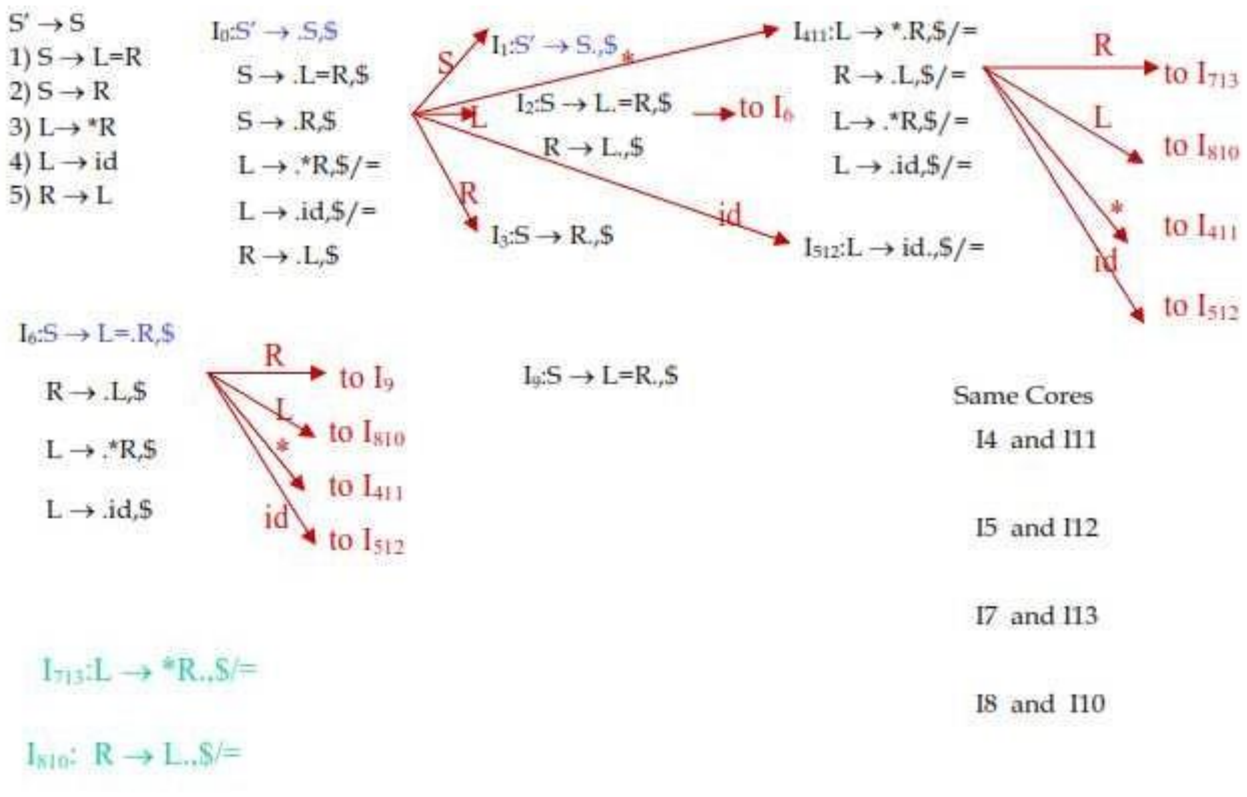
$\Downarrow$

I12: $A \rightarrow \alpha.,a/b$ - > reduce/reduce conflict

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

Example:

For the above Canonical LR Parsing table, we can get the following LALR(1) collection



Lecture #25

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**.

Example:

E → E+T | T

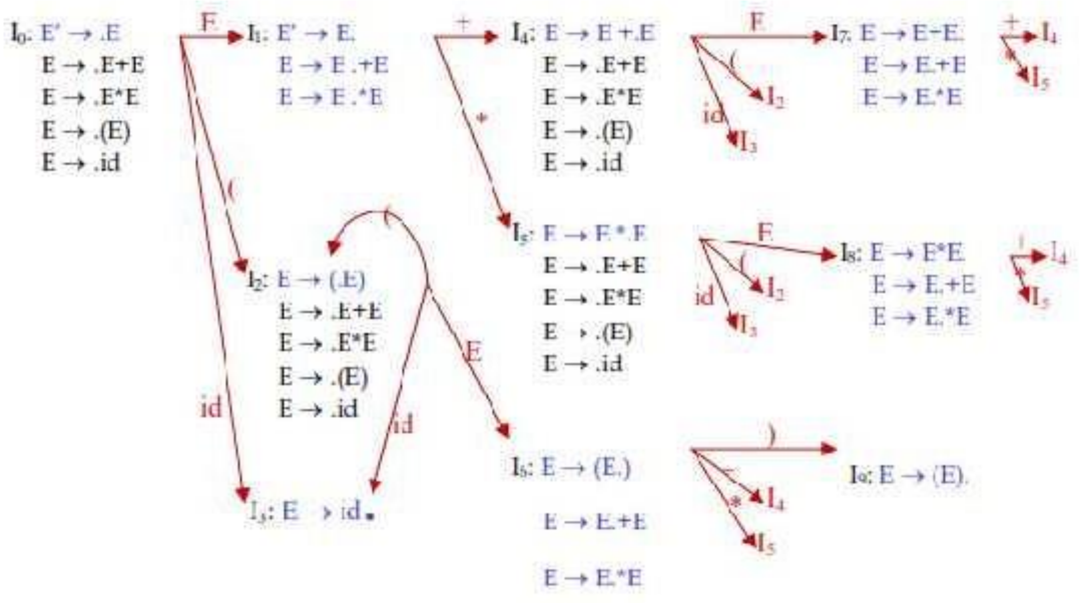
E → E+E | E*E |

(E) | id T → T*F

| F

F → (E) | id

Sets of LR(0) Items for Ambiguous Grammar



SLR-Parsing Tables for Ambiguous Grammar

FOLLOW(E) = { $,+,*,) }
State I7 has shift/reduce conflicts for symbols + and *.
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 *

State I8 has shift/reduce conflicts for symbols +
and *. 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 +

	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		

Module-2

Lecture #26

SYNTAX-DIRECTED TRANSLATION

• 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 any thing.
A string, a number, a memory location, a complex record.
• 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.

Example:

Production	**Semantic Rule**	**Program Fragment**
L → E **return**	print(E.val)	print(val[top-1])
E → E_1 + T	E.val = E_1.val + T.val	val[ntop] = val[top-2] + val[top]
E → T	E.val = T.val	
T → T_1 * F	T.val = T_1.val * F.val	val[ntop] = val[top-2] * val[top]
T → F	T.val = F.val	
F → (E)	F.val = E.val	val[ntop] = val[top-1]
F → **digit**	F.val = **digit**.lexval	val[top] = digit.lexval

• Symbols E, T, and F are associated with an attribute *val*.
• The token **digit** has an attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
• The *Program Fragment* above represents the implementation of the semantic rule for a bottom-up parser.
• 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*).
• The above model is suited for a desk calculator where the purpose is to evaluate and to generate code.

1. Intermediate Code Generation

• *Intermediate codes* are machine independent codes, but they are close to machine instructions.
• The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
• Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
– syntax trees can be used as an intermediate language.
– postfix notation can be used as an intermediate language.
– three-address code (Quadraples) can be used as an intermediate language

- we will use quadraples to discuss intermediate code generation
- quadraples are close to machine instructions, but they are not actual machine instructions.

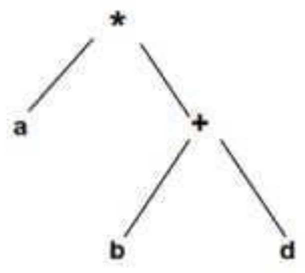
2 Syntax Tree

Syntax Tree is a variant of the Parse tree, where each leaf represents an operand and each interior node an operator.

Example:

<u>Production</u>	**<u>Semantic Rule</u>**		
E → E1 **op** E2	E.val = NODE (op, E1.val, E2.val)	E → (E1)	E.val = E1.val
E → - E1	E.val = UNARY (- , E1.val)	E → **id**	E.val = LEAF (id)

A sentence **a*(b+d)** would have the following syntax tree:



Postfix Notation

Postfix Notation is another useful form of intermediate code if the language is mostly expressions.

Example:

<u>Production</u>	<u>Semantic Rule</u>	<u>Program Fragment</u>
E → E1 **op** E2	E.code = E1.code || E2.code || op	print op
E → (E1)	E.code = E1.code	
E → **id**	E.code = id	print id

3 Three Address Code

• We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).

• The most general kind of three-address code is:

x **:=** y ***op*** z

where x, y and z are names, constants or compiler-generated temporaries; **op** is any operator.

• But we may also the following notation for quadraples (much better notation because it looks like a machine code instruction)

op y,z,x

apply operator op to y and z, and store the result in x.

4 Representation of three-address codes

Three-address code can be represented in various forms viz. Quadruples, Triples and Indirect Triples. These forms are demonstrated by way of an example below. Example: A = -B * (C + D) Three-Address code is as follows:

T1 = -B

T2 = C + D T3 = T1 * T2

A = T3

Quadruple:

	Operator	***Operand 1***	***Operand 2***	***Result***
(1)	-	B		T1
(2)	+	C	D	T2
(3)	*	T1	T2	T3
(4)	=	A	T3	

Triple:

	Operator	***Operand 1***	***Operand 2***
(1)	-	B	
(2)	+	C	D
(3)	*	(1)	(2)
(4)	=	A	(3)

Indirect Triple:

	Statement
(0)	(56)
(1)	(57)
(2)	(58)
(3)	(59)

	Operator	***Operand 1***	***Operand 2***
(56)	-	B	
(57)	+	C	D
(58)	*	(56)	(57)
(59)	=	A	(58)

Lecture #27

Translation of Assignment Statements

A statement A := - B * (C + D) has the following three-address translation: T1 := - B
T2 := C+D
T3 := T1* T2
A := T3

Production	Semantic Action
S → id := E	S.code = E.code || gen(id.place = E.place)

E → E1 + E2	E.place = newtemp();
	E.code = E1.code || E2.code || gen(E.place = E1.place + E2.place)

E → E1 * E2	E.place = newtemp();
	E.code = E1.code || E2.code || gen(E.place = E1.place * E2.place)

E → - E1	E.place = newtemp();
	E.code = E1.code || gen(E.place = - E1.place)

E → (E1)	E.place = E1.place; E.code = E1.code
E → id	E.place = id.place; E.code = null

1. Translation of Boolean Expressions

Grammar for Boolean Expressions is: E->E or E
E->E and E
E->not E E->(E) E->id
E->id relop id

There are two representations viz. Numerical and Control-Flow.

Numerical Representation of Boolean

- TRUE is denoted by 1 and FALSE by 0.
- Expressions are evaluated from left to right, in a manner similar to arithmetic expressions.

Example:
The translation for **A or B and C** is the three-address sequence:

T1 := B and C T2 := A or T1

Also, the translation of a relational expression such as A < B is the three-address sequence:

(1) if A < B goto (4) (2) T := 0
(3) goto (5) (4) T := 1 (5)

Therefore, a Boolean expression A < B or C can be translated as: (1) if A < B goto (4)
(2) T1 := 0
(3) goto (5) (4) T1 := 1
(5) T2 := T1 or C

Production	Semantic Action

E->E1 or E2	T = newtemp ();
	E.place = T;
	Gen (T = E1.place or E2.place)

E->E1 and E2	T = newtemp (); E.place = T; Gen (T = E1.place and E2.place)
E->not E1	T = newtemp (); E.place = T; Gen (T = not E1.place)
E → (E1)	E.place = E1.place; E.code = E1.code
E → id	E.place = id.place; E.code = null
E->id1 relop id2	T = newtemp (); E.place = T; Gen (if id1.place relop id2.place goto NEXTQUAD+3) Gen (T = 0) Gen (goto NEXTQUAD+2)
`	Gen (T = 1)

0 Quadruples are being generated and NEXTQUAD indicates the next available entry in the quadruple array.

Control-Flow Representation of Boolean Expressions

0 If we evaluate Boolean expressions by program position, we may be able to avoid evaluating the entire expressions.
0 In A or B, if we determine A to be true, we need not evaluate B and can declare the entire expression to be true.
0 In A and B, if we determine A to be false, we need not evaluate B and can declare the entire expression to be false.
0 A better code can thus be generated using the above properties.

Example:

The statement **if (A<B || C<D) x = y + z;** can be translated as
(1) if A<B goto (4) (2) if C<D goto (4) (3) goto (6)
(4) T = y + z
(5) X = T (6)

Here (4) is a true exit and (6) is a false exit of the Boolean expressions.

Lecture #28

Generating 3-address code for Numerical Representation of Boolean expressions

o Consider a production **E->E1 or E2** that represents the OR Boolean expression. If E1 is true, we know that E is true so we make the location TRUE for E1 be the same as TRUE for E. If E1 is false, then we must evaluate E2, so we make FALSE for E1 be the first statement in the code for E2. The TRUE and FALSE exits can be made the same as the TRUE and FALSE exits of E, respectively.

o Consider a production **E->E1 and E2** that represents the AND Boolean expression. If E1 is false, we know that E is false so we make the location FALSE for E1 be the same as FALSE for E. If E1 is true, then we must evaluate E2, so we make TRUE for E1 be the first statement in the code for E2. The TRUE and FALSE exits can be made the same as the TRUE and FALSE exits of E, respectively.

o Consider the production **E->not E** that represents the NOT Boolean expression. We may simply interchange the TRUE and FALSE exits of E1 to get the TRUE and FALSE exits of E.

o To generate quadruples in the manner suggested above, we use three functions- Makelist, Merge and Backpatch that shall work on the list of quadruples as suggested by their name.

o If we need to proceed to E2 after evaluating E1, we have an efficient way of doing this by modifying our grammar as follows:

E->E or M E
E->E and M E E->not E
E->(E) E->id
E->id relop id
M->ε

The translation scheme for this grammar would as follows:

Production	Semantic Action
E->E1 or M E2	BACKPATCH (E1.FALSE, M.QUAD); E.TRUE = MERGE (E1.TRUE, E2.TRUE); E.FALSE = E2.FALSE;
E->E1 and M E2	BACKPATCH (E1.TRUE, M.QUAD); E.TRUE = E2.TRUE; E.FALSE = MERGE (E1.FALSE, E2.FALSE);
E->not E1	E.TRUE = E1.FALSE; E.FALSE = E1.TRUE;

E->(E1)	E.TRUE = E1.TRUE; E.FALSE = E1.FALSE;
E->id	E.TRUE = MAKELIST (NEXTQUAD); E.FALSE = MAKELIST (NEXTQUAD + 1); GEN (if id.PLACE goto _); GEN (goto _);
E->id1 relop id2	E.TRUE = MAKELIST (NEXTQUAD); E.FALSE = MAKELIST (NEXTQUAD + 1); GEN (if id1.PLACE relop id2.PLACE goto _); GEN (goto _);
M->ε	M.QUAD = NEXTQUAD;

Example:

For the expression P<Q or R<S and T, the parsing steps and corresponding semantic actions are shown below. We assume that NEXTQUAD has an initial value of 100.

Step 1: P<Q gets reduced to E by E->id relop id. The grammatical form is E1 or R<S
and T.

We have the following code generated (Makelist).

```
100: if P<Q goto _
101: goto _
```

E1 is true if goto of 100 is reached and false if goto of 101 is reached.

Step 2: R<S gets reduced to E by E->id relop id. The grammatical form is E1 or E2 and
T.

We have the following code generated (Makelist).

```
102: if R<S goto _
103: goto _
```

E2 is true if goto of 102 is reached and false if goto of 103 is reached. Step 3: T gets reduced to E by E->id. The grammatical form is E1 or E2 and E3.
We have the following code generated (Makelist).

```
104: if T goto _
105: goto _
```

E3 is true if goto of 104 is reached and false if goto of 105 is reached.

Step 4: E2 and E3 gets reduced to E by E->E and E. The grammatical form is E1 or E4.

We have no new code generated but changes are made in the already generated code (Backpatch).

```
100: if P<Q goto _
101: goto _
102: if R<S goto 104
103: goto _
104: if T goto _
105: goto _
```

E4 is true only if E3.TRUE (goto of 104) is reached. E4 is false if E2.FALSE (goto of 103) or E3.FALSE (goto of 105) is reached (Merge).

Step 5: E1 or E4 gets reduced to E by E->E or E. The grammatical form is E.

We have no new code generated but changes are made in the already generated code (Backpatch).

```
100: if P<Q goto _
101: goto 102
102: if R<S goto 104
103: goto _
104: if T goto _
105: goto _
```

E is true only if E1.TRUE (goto of 100) or E2.TRUE (goto of 104) is reached (Merge). E is false if E4.FALSE (goto of 103 or 105) is reached.

1. Mixed Mode Expressions

- o Boolean expressions may in practice contain arithmetic sub expressions e.g. (A+B)>C.
- o We can accommodate such sub-expressions by adding the production E->E op E to our grammar.
- o We will also add a new field MODE for E. If E has been achieved after reduction using the above (arithmetic) production, we make E.MODE = arith, otherwise make E.MODE = bool.
- o If E.MODE = arith, we treat it arithmetically and use E.PLACE. If E.MODE = bool, we treat it as Boolean and use E.FALSE and E.TRUE.

Lecture #29
Statements that Alter Flow of Control

->In order to implement goto statements, we need to define a LABEL for a statement. A production can be added for this purpose:

S - > LABEL : S LABEL- > id

->The semantic action attached with this production is to record the LABEL and its value (NEXTQUAD) in the symbol table. It will also Backpatch any previous references to this LABEL with its current value.

->Following grammar can be used to incorporate structured Flow-of-control constructs: (1) S->if E then S
(2) S->if E then S else S
(3) S->while E do S (4) S->begin L end
(5) S->A
(6) L->L ; S (7) L->S

Here, S denotes a statement, L a statement-list, A an assignment statement and E a Boolean-valued expression.

1. Translation Scheme for statements that alter flow of control

->We introduce a new field NEXT for S and L like TRUE and FALSE for E. S.NEXT and L.NEXT are respectively the pointers to a list of all conditional and unconditional jumps to the quadruple following statement S and statement-list L in execution order.
->We also introduce the marker non-terminal M as in the case of grammar for Boolean expressions. This is put before statement in if-then, before both statements in if-then-else and the statement in while-do as we may need to proceed to them after evaluating E. In case of while-do, we also need to put M before E as we may need to come back to it after executing S.
->In case of if-then-else, if we evaluate E to be true, first S will be executed. After this we should ensure that instead of second S, the code after this if-then-else statement be executed. We thus place another non-terminal marker N after first S i.e. before else.
The grammar now is as follows:

(1) S->if E then M S
(2) S->if E then M S N else M S (3) S->while M E do M S
(4) S->begin L end
(5) S->A
(6) L->L ; M S (7) L->S
(8) M->ε
(9) N->ε

The translation scheme for this grammar would as follows:

Production	Semantic Action
S->if E then M S1	BACKPATCH (E.TRUE, M.QUAD) S.NEXT = MERGE (E.FALSE, S1.NEXT)
S->if E then M1 S1 N else M2 S2	BACKPATCH (E.TRUE, M1.QUAD) BACKPATCH (E.FALSE, M2.QUAD) S.NEXT = MERGE (S1.NEXT, N.NEXT, S2.NEXT)
S->while M1 E do M2 S1	BACKPATCH (S1.NEXT, M1.QUAD) BACKPATCH (E.TRUE, M2.QUAD) S.NEXT = E.FALSE GEN (goto M1.QUAD)
S->begin L end	S.NEXT = L.NEXT
S->A	S.NEXT = MAKELIST ( )
L->L1 ; M S	BACKPATCH (L1.NEXT, M.QUAD) L.NEXT = S.NEXT
L->S	L.NEXT = S.NEXT
M->ε	M.QUAD = NEXTQUAD
N->ε	N.NEXT = MAKELIST (NEXTQUAD) GEN (goto _)

Lecture #30
Postfix Translations

In an production A->α, the translation rule of A.CODE consists of the concatenation of the CODE translations of the non-terminals in α in the same order as the non-terminals appear in α. Productions can be factored to achieve Postfix form.

1. Postfix translation of while statement

The production

S->while M1 E do M2 S1 can be factored as
S->C S1
C->W E do
W->while

A suitable translation scheme would be

Production	Semantic Action
W->while	W.QUAD = NEXTQUAD
C->W E do	C.QUAD = W.QUAD BACKPATCH (E.TRUE, NEXTQUAD) C.FALSE = E.FALSE
S->C S1	BACKPATCH (S1.NEXT, C.QUAD) S.NEXT = C.FALSE GEN (goto C.QUAD)

2. Postfix translation of for statement

Consider the following production which stands for the for-statement

S->for L = E1 step E2 to E3 do S1

Here L is any expression with l-value, usually a variable, called the index. E1, E2 and E3 are expressions called the initial value, increment and limit, respectively. Semantically, the for statement is equivalent to the following program.

```
Begin

INDEX = addr ( L );
*INDEX = E1; INCR = E2; LIMIT = E3;
while *INDEX <= LIMIT do begin

end

end

code for statement S1;
*INDEX = *INDEX + INCR;
```

The non-terminals L, E1, E2, E3 and S appear in the same order as in the production. The production can be factored as

(1) F->for L
(2) T->F = E1 by E2 to E3 do
(3) S->T S1

A suitable translation scheme would be

Production	Semantic Action
F->for L	F.INDEX = L.INDEX

T->F = E1 by E2 to E3 do	GEN (*F.INDEX = E1.PLACE) INCR = NEWTEMP ( )
	LIMIT = NEWTEMP ( ) GEN (INCR = E2.PLACE)
	GEN (LIMIT = E3.PLACE)
	T.QUAD = NEXTQUAD
	T.NEXT = MAKELIST (NEXTQUAD) GEN (IF *F.INDEX >
	LIMIT goto _) T.INDEX = F.INDEX
	T.INCR = INCR

S->T S1	BACKPATCH (S1.NEXT, NEXTQUAD) GEN (*T.INDEX = *T.INDEX + T.INCR) GEN (goto T.QUAD)
S.NEXT = T.NEXT	

Lecture #31
Array references in arithmetic expressions

Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive locations.
For a one-dimensional array A:

Base (A) is the address of the first location of the array A,
width is the width of each array element.
low is the index of the first array element

location of A[i] = baseA+(i-low)*width baseA+(i-low)*width
can be re-written as

i*width + (baseA-low*width)

should be computed at run-time   can be computed at compile-time

So, the location of A[i] can be computed at the run-time by evaluating the formula i*width+c where c is (baseA-low*width) which is evaluated at compile-time.

Intermediate code generator should produce the code to evaluate this formula i*width+c (one multiplication and one addition operation).
A two-dimensional array can be stored in either row-major (row-by-row) or column-major (column-by-column).

Most of the programming languages use row-major method.

The location of A[i1,i2] is baseA+ ((i1-low1)*n2+i2-low2)*width

baseA is the location of the array A.
low1 is the index of the first row
low2 is the index of the first column
n2 is the number of elements in each row
width is the width of each array element

Again, this formula can be re-written as

((i1*n2)+i2)*width + (baseA-((low1*n1)+low2)*width)

should be computed at run-time   can be computed at compile-time

Arrays of any dimension can be dealt in a similar but general manner.

In general, the location of A[i1,i2,...,ik] is

((... ((i1*n2)+i2) ...)*nk+ik)*width + (baseA- ((...((low1*n1)+low2)...)*nk+lowk)*width)

So, the intermediate code generator should produce the codes to evaluate the following formula (to find the location of A[i1,i2,...,ik]) :

((... ((i1*n2)+i2) ...)*nk+ik)*width + c

To evaluate the ((... ((i1*n2)+i2) ...)*nk+ik portion of this formula, we can use the recurrence equation:

e1 = i1
em = em-1 * nm + im

1. Grammar and Translation Scheme

The grammar and suitable translation scheme for arithmetic expressions with array references is as given below:

Production	Semantic Action

S → L = E	if (L.OFFSET = NULL) then GEN (L.PLACE = E.PLACE)
	else GEN(L.PLACE [L.OFFSET] = E.PLACE)

E → E1 + E2	E.PLACE = NEWTEMP ( )
	GEN (E.PLACE = E1.PLACE + E2.PLACE) E → (E1)
	E.PLACE = E1.PLACE

E → L	if (L.OFFSET = NULL) then E.PLACE = L.PLACE
	else {E.PLACE = NEWTEMP ( ); GEN (E.PLACE = L.PLACE[L.OFFSET])}

L → id	L.PLACE = id.PLACE L.OFFSET = NULL
L → ELIST]	L.PLACE = NEWTEMP( ) L.OFFSET = NEWTEMP ( ) GEN (L.PLACE = ELIST.ARRAY - C) GEN (L.OFFSET = ELIST.PLACE * WIDTH (ELIST.ARRAY))
ELIST → ELIST1 , E	ELIST.ARRAY = ELIST1.ARRAY ELIST.PLACE = NEWTEMP ( ) ELIST.NDIM = ELIST1.NDIM + 1 GEN (ELIST.PLACE = ELIST1.PLACE * LIMIT (ELIST.ARRAY, ELIST.NDIM)) GEN (ELIST.PLACE = E.PLACE + ELIST.PLACE)
ELIST → id [E	ELIST.ARRAY = id.PLACE ELIST.PLACE = E.PLACE E = 1

Here, NDIM denotes the number of dimensions, LIMIT (AARAY, i) function returns the upper limit along the ith dimension of ARRAY i.e. ni, WIDTH (ARRAY) returns the number of bytes for one element of ARRAY.

2. Declarations

Following is the grammar and a suitable translation scheme for declaration statements:

<u>Production</u>	<u>Semantic Action</u>
D->integer, id	ENTER (id.PLACE, integer) D.ATTR = integer
D->real, id	ENTER (id.PLACE, real) D.ATTR = real
D->D1, id	ENTER (id.PLACE, D1.ATTR) D.ATTR = D1.ATTR

Here, ENTER makes the entry into symbol table while ATTR is used to trace the data type.

3. Procedure Calls

Following is the grammar and a suitable translation scheme for Procedure Calls:

<u>Production</u>	<u>Semantic Action</u>
S->call id (ELIST)	for each item p on QUEUE do
GEN (param p)	GEN (call id.PLACE)
ELIST->ELIST, E	append E.PLACE to the end of QUEUE
ELIST->E	initialize QUEUE to contain only E.PLACE QUEUE is used to store

the list of parameters in the procedure call.

4. Case Statements

The case statement has following syntax:

```
switch E
begin
end
case V1: S1 case V2: S2
.
case Vn-1: Sn-1 default: Sn The translation scheme for this shown below:
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
```

Lecture #32
SYMBOL TABLES

- Symbol table is a data structure meant to collect information about names appearing in the source program.
- It keeps track about the scope/binding information about names.
- Each entry in the symbol table has a pair of the form (name and information).
- Information consists of attributes (e.g. type, location) depending on the language.
- Whenever a name is encountered, it is checked in the symbol table to see if already occurs. If not, a new entry is created.
- In some cases, the symbol table record is created by the lexical analyzer as soon as the name is encountered in the input, and the attributes of the name are entered when the declarations are processed.
- If same name can be used to denote different program elements in the same block, the symbol table record is created only when the name's syntactic role is discovered.

Operations on a Symbol Table

- Determine whether a given name is in the table
- Add a new name to the table
- Access information associated to a given name
- Add new information for a given name
- Delete a name (or a group of names) from the table

Implementation

- Each entry in a symbol table can be implemented as a record that consists of several fields.
- The entries in symbol table records are not uniform and depend on the program element identified by the name.
- Some information about the name may be kept outside of the symbol table record and/or some fields of the record may be left vacant for the reason of uniformity. A pointer to this information may be stored in the record.
- The name may be stored in the symbol table record itself, or it can be stored in a separate array of characters and a pointer to it in the symbol table.
- The information about runtime storage location, to be used at the time of code generation, is kept in the symbol table.
- There are various approaches to symbol table organization e.g. Linear List, Search Tree and Hash Table.

Linear List

- It is the simplest approach in symbol table organization.
- The new names are added to the table in the order they arrive.
- A name is searched for its existence linearly.
- The average number of comparisons required are proportional to 0.5*(n+1) where n=number of entries in the table.
- It takes less space but more access time.

Search Tree

- It is more efficient than Linear Trees.
- We provide two links- left and right, which point to record in the search tree.

- A new name is added at a proper location in the tree such that it can be accessed alphabetically.
- For any node name1 in the tree, all names accessible by following the left link precede name1 alphabetically.
- Similarly, for any node name1 in the tree, all names accessible by following the right link succeed name1 alphabetically.
- The time for adding/searching a name is proportional to (m+n) $\log_2 n$.

Hash Table

- A hash table is a table of k-pointers from 0 to k-1 that point to the symbol table and record within the symbol table.
- To search a value, we find out the hash value of the name by applying suitable hash function.
- The hash function maps the name into an integer value between 0 and k-1 and uses it as an index in the hash table to search the list of the table records that are built on that hash index.
- To add a non-existent name, we create a record for that name and insert it at the head of the list.

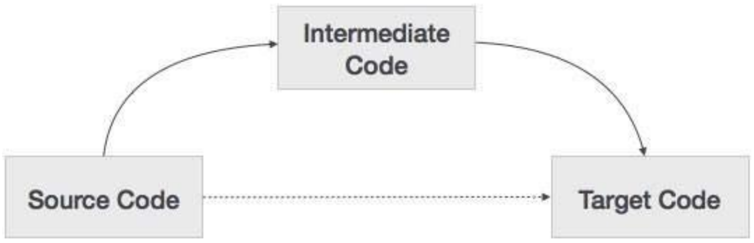
32.3 Scope Information

- Each name possesses a region of validity within the source program called the scope of that name.
- The rules governing the scope of names in a block-structured language are as follows:
- A name declared within block B is valid only within B.
- If block B1 is nested within B2, then any name that is valid for B2 is also valid for B1, unless identifier for that name is re-declared in B1.
- These rules require a more complicated symbol table organization that simply a list of associations between names and attributes.
- One technique is to keep multiple symbol tables for each active block:
- Each table is list of names and their associated attributes, and the tables are organized on stack.
- Whenever a new block is entered, a new table is pushed on the stack.
- When a declaration is compiled, the table on the stack is searched for the name.
- If name is not found it is inserted.
- When a reference is translated, it is searched in all tables starting from top.
- Another technique is to represent scope information in the symbol table.
- Store the nesting depth of each procedure block in the symbol table.
- Use the (procedure name, nesting depth) pair as the key to accessing the information from the table.
- The nesting depth of a procedure is a number that is obtained by starting with a value of one for the main and adding one to it every time we go from an enclosing to an enclosed procedure. It counts the number of procedure in the referencing environment of a procedure.

Lecture#33

Intermediate Code Generation

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
- **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;

r2 = b + r1;

a = r2
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg_1	arg_2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Declarations

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

Example:

We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

```
int a;

float b;

Allocation process:

{offset = 0}

   int a;

   id.type = int

   id.width = 2


offset = offset + id.width

{offset = 2}

   float b;

   id.type = float

   id.width = 4


offset = offset + id.width

{offset = 6}
```

To enter this detail in a symbol table, a procedure *enter* can be used. This method may have the following structure:

```
enter(name, type, offset)
```

This procedure should create an entry in the symbol table, for variable *name*, having its type set to type and relative address *offset* in its data area.

Lecture #34

Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

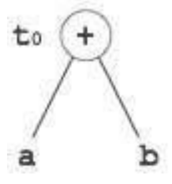
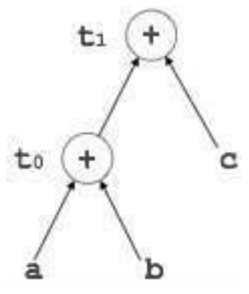
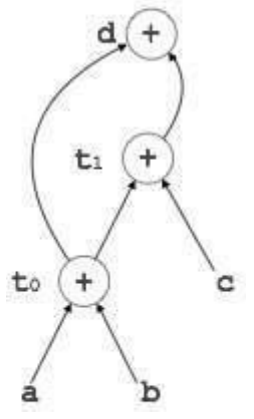
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

$t_0 = a + b$

$t_1 = t_0 + c$

$d = t_0 + t_1$

 [$t_0 = a + b$]	 [$t_1 = t_0 + c$]	 [$d = t_0 + t_1$]

Boolean Expressions and Control Flow

BOOLEAN EXPRESSIONS are constructed using boolean operators. We will consider here the following rules.

$E \longmapsto$	E **or** E
$E \longmapsto$	E **and** E

$E \longmapsto \textbf{not}\ E$
$E \longmapsto (\ E\ )$
$E \longmapsto \textbf{id}\ \textbf{relop}\ \textbf{id}$
$E \longmapsto \textbf{true}$
$E \longmapsto \textbf{false}$

- Boolean expressions are used as conditions for statements changing the flow of control.
- Evaluation of boolean expressions can be optimized if it is sufficient to evaluate a part of the expression that determines its value.
- When translating Boolean expressions into three-address code, we can use two different methods.

  Numerical method.
  Assign numerical values to **true** and **false** and evaluate the expression analogously to an arithmetic expression. This is convenient for boolean expressions which are not involved in flow of control constructs.
  Jump method.
  Evaluate a boolean expression *E* as a sequence of conditional and unconditional jumps to location *E*.*true* (if *E* is **true**) or to *E*.*false*. To be detailed shortly.

WHILE STATEMENTS WITH THE NUMERICAL METHOD.

The following syntax-directed translation generates code for a while statement.

Production	Semantic Rule
$S \longmapsto \textbf{while}\ E\ \textbf{repeat}\ S_1$	*S.begin* := **newlabel**
	S.after := **newlabel**
	$code_1$:= *generate*(*S.begin* '**:**') | | *E.code* | |
	$code_2$:= *generate*('**if** ' *E.place* '**= 0** ' '**goto**' *S.after*) | | S_1.*code*
	$code_3$:= *generate*('**goto**' *S.begin*) | | *generate*(*S.after*)
	S.code := $code_1$ | | $code_2$ | | $code_3$

With respect to the previous syntax-directed translation

- we have added two new synthesized attributes *S.begin* and *S.after*.
- When the value of *E* becomes zero, control leaves the while statement

Lecture #35

FLOW OF CONTROL STATEMENTS WITH THE JUMP METHOD.

We will consider here the following rules.

S	$\longmapsto$	$\textbf{if}\ E\ \textbf{then}\ S_1$
S	$\longmapsto$	$\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2$
S	$\longmapsto$	$\textbf{while}\ E\ \textbf{repeat}\ S_1$

- In each of these productions, *E* is the boolean expression to be translated.
- The boolean expression *E* is associated with two labels (that are **inherited** attributes in the following semantic rules)
    - *E*.*true* the label to which control flows if *E* is **true**,
    - *E*.*false* the label to which control flows if *E* is **false**.
- In each of these productions, *S* is a flow of control statement associated with two attributes
    - *S*.*next* which is a label that is attached to the first 3-address statement to be executed after the code for *S* , *S*.*next* is an **inherited** attribute,
    - *S*.*code* is the translation code for *S*, as usual it is a **synthesized** attribute.

Production	Semantic Rule
$S \longmapsto \textbf{if}\ E\ \textbf{then}\ S_1$	*E.true* := *newlabel*
	E.false := *S.next*
	S_1*.next* := *S.next*
	S.code := *E.code* | | *generate*(*E.true* '**:**') | | S_1*.code*
$S \longmapsto \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2$	*E.true* := *newlabel*
	E.false := *newlabel*
	S_1*.next* := *S.next*
	S_2*.next* := *S.next*
	code$_1$:= *E.code* | | *generate*(*E.true* '**:**') | | S_1*.code*
	code$_2$:= *generate*('**goto**' *S.next*) | |
	code$_3$:= *generate*(*E.false* '**:**') | | S_2*.code*
	S.code := *code*$_1$ | | *code*$_2$ | | *code*$_3$
$S \longmapsto \textbf{while}\ E\ \textbf{repeat}\ S_1$	*S.begin* := *newlabel*
	E.true := *newlabel*
	E.false := *S.next*
	S_1*.next* := *S.begin*

	$code_1$:= *generate*(*S.begin* '**:**') | | *E.code*
	$code_2$:= *generate*(*E.true* '**:**') | | S_1.*code*
	$code_3$:= *generate*('**goto**' *S.begin*)
	S.code := $code_1$ | | $code_2$ | | $code_3$

Warning! The above is a syntax-directed definition: It provides formulas for the computation of the attributes *S.code* (via the computations of the other attributes).

- Since several attributes are inherited and since each action above appears after its associated production, **this is not a translation scheme**.
- However it is an *L*-attributed definition.
- Then its conversion into a translation scheme is obvious.
- From now on, we may present a translation scheme as a syntax-directed definition if the latter is an *L*-attributed definition.
- The reason is to make large translation schemes easier to read.

TRANSLATION OF BOOLEAN EXPRESSIONS WITH THE JUMP METHOD. We will consider here the following rules.

E	$\longmapsto$	E_1 **or** E_2
E	$\longmapsto$	E_1 **and** E_2
E	$\longmapsto$	**not** E_1
E	$\longmapsto$	(E_1)
E	$\longmapsto$	$\mathbf{id_1}$ **relop** $\mathbf{id_2}$
E	$\longmapsto$	**true**
E	$\longmapsto$	**false**

- Here again each symbol *E* is associated with two inherited attributes
  - *E.true* the label to which control flows if *E* is **true**,
  - *E.false* the label to which control flows if *E* is **false**.
- The attributes *E.true* and *E.false* of *E* will be definned when the flow of control (where *E* appears) is translated.

Production	Semantic Rule
E $\longmapsto$ E_1 **or** E_2	E_1.*true* := *E.true*
	E_1.*false* := *newlabel*
	E_2.*true* := *E.true*
	E_2.*false* := *E.false*

	E.*code* := *E*₁.*code* | | *generate*(*E*₁.*false*'**:**') | | *E*₂.*code*
$E \longmapsto E_1$ **and** E_2	*E*₁.*true* := *newlabel*
	*E*₁.*false* := *E*.*false*
	*E*₂.*true* := *E*.*true*
	*E*₂.*false* := *E*.*false*
	E.*code* := *E*₁.*code* | | *generate*(*E*₁.*true*'**:**') | | *E*₂.*code*
$E \longmapsto$ **not** E_1	*E*₁.*true* := *E*.*false*
	*E*₁.*false* := *E*.*true*
	E.*code* := *E*₁.*code*
$E \longmapsto (E_1)$	*E*₁.*true* := *E*.*true*
	*E*₁.*false* := *E*.*false*
	E.*code* := *E*₁.*code*
$E \longmapsto \mathbf{id_1}\ \mathbf{relop}\ \mathbf{id_2}$	*code*₁ := *generate*('**if**' $\mathbf{id_1}$.*place* $\mathbf{relop}\ \mathbf{id_2}$.*place* '**goto**' *E*.*true*)
	*code*₂ := *generate*('**goto**' *E*.*false*)
	E.*code* := *code*₁ | | *code*₂
$E \longmapsto$ **true**	*E*.*code* := *generate*('**goto**' *E*.*true*)
$E \longmapsto$ **false**	*E*.*code* := *generate*('**goto**' *E*.*false*)

Example 5 *Let us consider the expression*
a < b
Assume that

- the attributes *true* and *false* exist for the entire expression
- as labels Ltrue and Lfalse respectively.

Then the translation is

```
if a < b goto Ltrue
goto Lfalse
```

Observations.

- Of course, this is not optimal and looks funny since the expression to translate is a sentence of the target language!
- But this *jump method* allows us to translate more involved expressions (which are not part of the target language) like those of the following examples.

Example 6 *Now consider the expression*
a < b or c < d
Again assume that the labels Ltrue and Lfalse have been set for the entire expression. Then the translation is

```
    if a < b goto Ltrue
    goto L1
L1: if c < d goto Ltrue
    goto Lfalse
```

Example 7 *Now consider the expression*
a < b or (c < d and e < f)
Then the translation is

```
    if a < b goto Ltrue
    goto L1
L1: if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse
```

Of course the generated code is not optimal! Indeed the second statement can be eliminated.

Example 8 *Finally consider the expression*

```
while a < b do
   if c < d then
      x := y + z
   else
      x := y - z
```

Then the translation is

```
L1:     if a < b goto L2
        goto Lnext
L2:     if c < d goto L3
        goto L4
L3:     t1 := y + z
        x := t1
        goto L1
L4:     t2 := y - z
        x := t2
        goto L1
Lnext:
```

Lecture #36

Backpatching

A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump. For example, the translation of the boolean expression B in if (B) S contains a jump, for when B is false, to the instruction following the code for S. In a one-pass translation, B must be translated before S is examined. Labels can be passed as inherited attributes to where the relevant jump instructions were generated. But a separate pass is then needed to bind labels to addresses.In backpatching, lists of jumps are passed as synthesized attributes. Specifically, when a jump is generated, the target of the jump is temporarily left unspecified. Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target label.

One-Pass Code Generation Using Backpatching

Backpatching can be used to generate code for boolean expressions and flow-of-control statements in one pass. Synthesized attributes truelist and falselist of nonterminal B are used to manage labels in jumping code for boolean expressions. B. truelist will be a list of jump or conditional jump instructions into which we must insert the label to which control goes if B is true. B.falselist likewise is the list of instructions that eventually get the label to which control goes when B is false. As code is generated for B, jumps to the true and false exits are left incomplete, with the label field unfilled. These incomplete jumps are placed on lists pointed to by B.truelist and B.falselist, as appropriate. A statement S has a synthesized attribute S.nextlist, denoting a list of jumps to the instruction immediately following the code for S.

Instructions are generated into an instruction array, and labels will be indices into this array. To manipulate lists of jumps, we use three functions: makelist(i), merge(p1,p2),backpatch(p,i)
• makelist(i) creates a new list containing only i, an index into the array of instructions; makelist returns a pointer to the newly created list.

• merge(p1,p2) concatenates the lists pointed to 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 instructions on the list pointed to by p. Backpatching for Boolean Expressions

$B \rightarrow B1 \mid\mid M\ B2 \ \mid\ B1\ \&\&\ M\ B2 \ \mid\ !\ B \ \mid\ (\ B\ ) \mid E_1 \text{ rel } E_2 \mid \text{true} \mid \text{false}$

$M \rightarrow \varepsilon$

- A marker nonterminal *M* in the grammar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated.

Translation Scheme for Boolean Expressions

$B \rightarrow B1 \mid\mid M\ B2$ { *backpatch(B1.falselist, M.instr);*
B.truelist = merge(B1. truelist, B2.truelist);
B.falselist = B2.falselist; }

$B \rightarrow B1\ \&\&\ M\ B2$ { *backpatch(B1.truelist, M.instr);*
B.truelist = B2.truelist;
B.falselist = merge(B1.falselist,B2.falselist); }

$B \rightarrow !\ B1$ { *B.truelist = B1.falselist;*
B.falselist = Bi.truelist; }

$B \rightarrow (\ B1\ )$ { *B.truelist = B1.truelist;*
B.falselist = B1.falselist; }

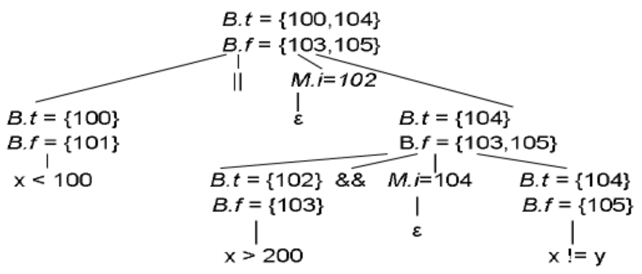
$B \rightarrow E1$ **rel** $E2$ { *B.truelist = makelist(nextinstr);*
B.falselist = makelist(nextinstr + 1);
emit('**if**' *E1.addr* **rel.***op E2.addr* 'goto _');
emit('goto _'); }

$B \rightarrow$ **true** { *B.truelist = makelist(nextinstr); emit('goto _');* }

$B \rightarrow$ **false** { *B.falselist = makelist(nextinstr); emit('goto _');* }

$M \rightarrow \varepsilon$ { *M.instr = nextinstr,* }

Applying the above SDT to the statement **if (x < 100 II x > 200 && x != y) x = 0;** gives a annotated parse tree as below.



Backpatching for Flow-Of-Control Statements

The grammar is given by the following productions :

$S \rightarrow$ **if** $(B)\ S$ | **if** $(B)\ S$ **else** S | **while** $(B)\ S$
    | $\{ L \}$ | A ;
$L \rightarrow L\ S$ | S

Where S – statement
L – list of statements
A- assignment statement
B – boolean expression

Translation of flow-of-control Statements Using Backpatching

S →if (B) M S1
{ backpatch{B.truelist, M.instr);
S.nextlist = merge(B.falselist, S1.nextlist); }

S → if (B) M1 S1 N else M2 S2
{ backpatch(B .truelist, M1.instr);
backpatch(B .falselist, M2.instr);
temp = merge(S1.nextlist, N.nextlist);
S.nextlist = merge(temp, S2.nextlist); }

S → while M1 (B) M2 S1
{ backpatch(S1.nextlist, M1.instr);
backpatch(B. truelist, M2.instr);
S.nextlist = B. falselist;
emit('goto' M1.instr); }

S → { L } *{ S.nextlist = L.nextlist; }*

S →A ; *{ S.nextlist =* ***null;*** *}*
M →ε *{ M.instr = nextinstr; }*
S → { L }
{ N.nextlist = makelist(nextinstr);
emit('goto _'); }

L → L1 M S
{ backpatch(L1.nextlist, M.instr)
L. Nextlist = S. nextlist; }

L → S *{ L. nextlist — S. nextlist; }*

Intermediate Code for Procedures

Assume that parameters are passed by value. Suppose that a is an array of integers, and that f is a function from integers to integers. Then, the assignment n = f (a [i]) ;might translate into the following three-address code:

```
1. t 1 = i * 4
2. t 2 = a [ t 1 ]
3. param t2
4. t3 = call f, 1
5. n = t 3
```

The grammar below adds functions to the source language

$D \rightarrow \textit{define } T \textit{ id } (F) \{ S \}$

$F \rightarrow \varepsilon \mid T \textit{ id } , F$

$S \rightarrow \textit{return } E ;$

$E \rightarrow \textit{id } (A)$

$A \rightarrow \varepsilon \mid E , A$

Module-3:

Lecture #37

RUN TIME ADMINISTRATION

- The places of the data objects that can be determined at compile time will be *allocated statically*.
- But the places for the some of data objects will be *allocated at run-time*.

  The allocation of de-allocation of the data objects is managed by the *run-time support package*.

  →Run-time support package is loaded together with the generate target code.

  →The structure of the run-time support package depends on the semantics of the programming language (especially the semantics of procedures in that language).

Procedure Activations

- Each execution of a procedure is called as *activation of that procedure*.
- An execution of a procedure starts at the beginning of the procedure body;
- When the procedure is completed, it returns the control to the point immediately after the place where that procedure is called.
- Each execution of a procedure is called as its *activation*.
- *Lifetime* of an activation of a procedure is the sequence of the steps between the first and the last steps in the execution of that procedure (including the other procedures called by that procedure).
- If a and b are procedure activations, then their lifetimes are either non-overlapping or are nested.
- If a procedure is recursive, a new activation can begin before an earlier activation of the same procedure has ended.

Activation Tree

- We can use a tree (called activation tree) to show the way control enters and leaves activations.

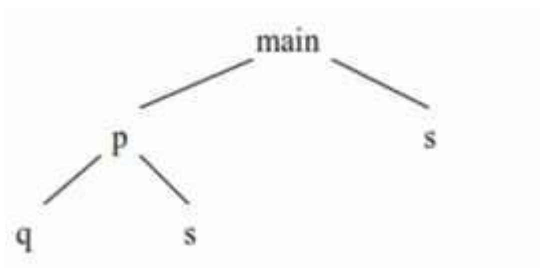
- In an activation tree:
  - Each node represents an activation of a procedure.
  - The root represents the activation of the main program.
  - The node a is a parent of the node b iff the control flows from a to b.
  - The node a is left to to the node b iff the lifetime of a occurs before the lifetime of b.

Example:

```
program main;                    enter main
procedure s;                     enter p
begin ... end;                   enter q
procedure p;                     exit q
procedure q;                     enter s
```

begin ... end;
begin q; s; end;
begin p; s; end;

exit s
exit p
enter s
exit s
exit main



Control Stack

- The flow of the control in a program corresponds to a depth-first traversal of the activation tree that:
  - starts at the root,
  - visits a node before its children, and
  - recursively visits children at each node an a left-to-right order.
- A stack (called **control stack**) can be used to keep track of live procedure activations.
  - An activation record is pushed onto the control stack as the activation starts.
  - That activation record is popped when that activation ends.
- When node n is at the top of the control stack, the stack contains the nodes along the path from n to the root.

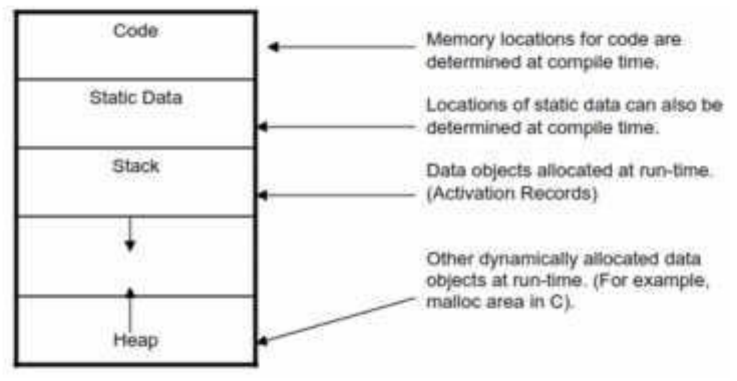
Variable Scopes

- The same variable name can be used in the different parts of the program.
- The scope rules of the language determine which declaration of a name applies when the name appears in the program.
- An occurrence of a variable (a name) is:
  - **local**: If that occurrence is in the same procedure in which that name is declared.
  - **non-local**: Otherwise (ie. it is declared outside of that procedure) Example:

```
procedure p;
var b:real;
procedure p;
var a: integer;                              a  is  local
begin a := 1;  b := 2; end;                  b  is  non-local
begin ... end;
```

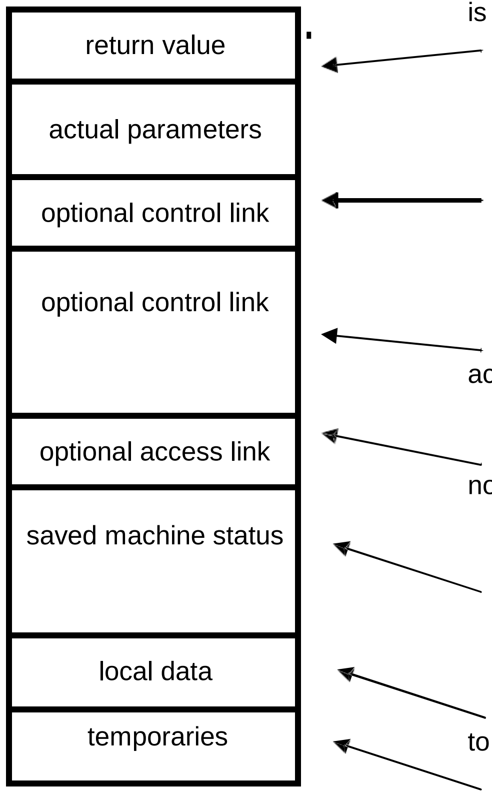
Lecture #38

Storage Organization


Code
Static Data
Stack
Heap
Memory locations for code are determined at compile time.
Locations of static data can also be determined at compile time.
Data objects allocated at run-time. (Activation Records)
Other dynamically allocated data objects at run-time. (For example, malloc area in C).

Activation Records

• Information needed by a single execution of a procedure is managed using a contiguous block of storage called **activation record**.
• An activation record is allocated when a procedure is entered, and it is de-allocated when that procedure exited.
• Size of each field can be determined at compile time (Although actual location of the activation record is determined at run-time).
– Except that if the procedure has a local variable and its size depends on a parameter, its size is determined at the run time.



The returned value of the called procedure is returned in this field to the calling procedure. In practice, we may use a machine register for the return value.

The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.

The optional control link points to the activation record of the caller.

The optional access link is used to refer to nonlocal data held in other activation records.

The field for saved machine status holds information about the state of the machine before the procedure is called.

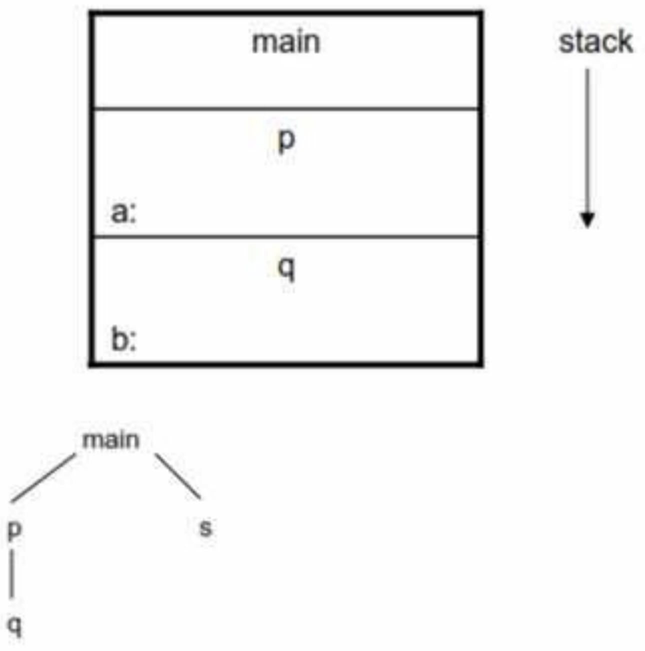
The field of local data holds data that local to an execution of a procedure..

Temporay variables is stored in the field of temporaries.

Example:

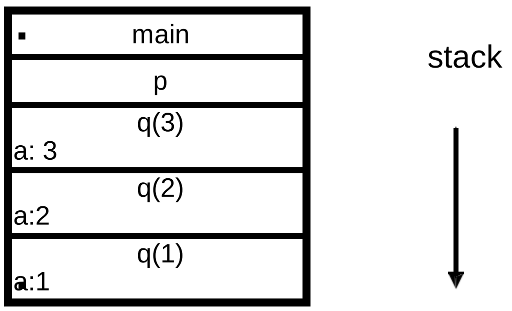
(For a non-recursive procedure)

```
program main;
  procedure p;
    var a:real;
    procedure q;
      var b:integer;
      begin ... end;
    begin q; end;
  procedure s;
    var c:integer;
    begin ... end;
  begin p; s; end;
```



(For a recursive procedure)

```
program main; procedure p; function
q(a:integer):integer;
begin
if (a=1) then q:=1; else q:=a+q(a-1);
end;
begin q(3); end;
begin p; end;
```



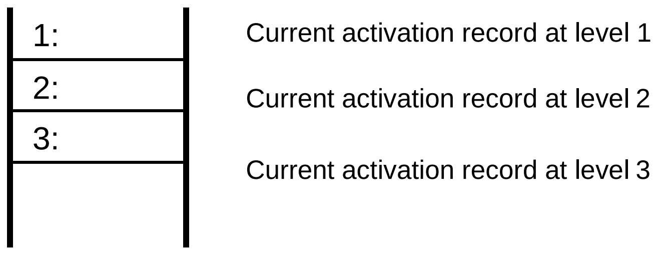
Creation of Activation Records

- Who allocates an activation record of a procedure?
  - Some part of the activation record of a procedure is created by that procedure immediately after that procedure is entered.
  - Some part is created by the caller of that procedure before that procedure is entered.

- Who deallocates?
  - Callee de-allocates the part allocated by Callee.
  - Caller de-allocates the part allocated by Caller.

Displays

- An array of pointers to activation records can be used to access activation records.
- This array is called as displays.
- For each level, there will be an array entry.


1:
2:
3:
Current activation record at level 1
Current activation record at level 2
Current activation record at level 3

Lecture #39

ERROR DETECTION AND RECOVERY

- 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 recover 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 change 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.

Lecture #40

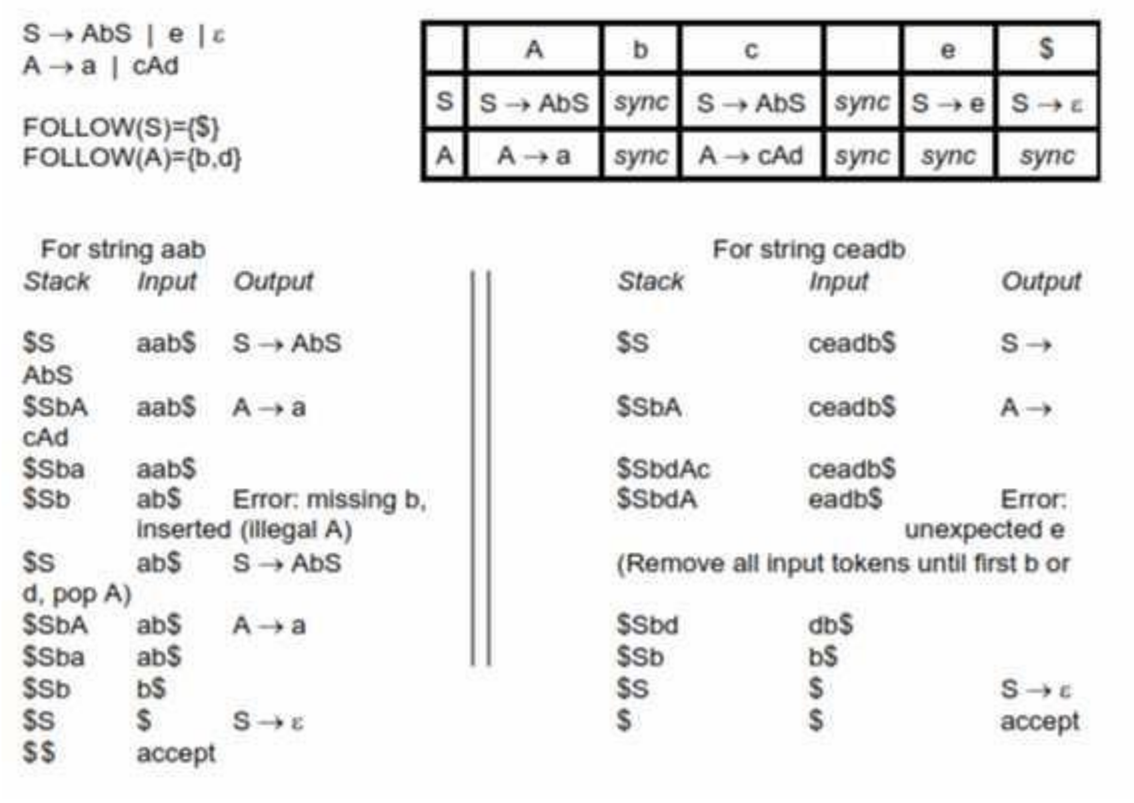
Error Recovery in Predictive Parsing

- 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.

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.

Example:



$S \rightarrow AbS \mid e \mid \varepsilon$
$A \rightarrow a \mid cAd$

FOLLOW(S)={$}
FOLLOW(A)={b,d}

	A	b	c		e	$
S	$S \rightarrow AbS$	*sync*	$S \rightarrow AbS$	*sync*	$S \rightarrow e$	$S \rightarrow \varepsilon$
A	$A \rightarrow a$	*sync*	$A \rightarrow cAd$	*sync*	*sync*	*sync*

For string aab

```
Stack     Input    Output

$S        aab$     S → AbS
$SbA      aab$     A → a
$Sba      aab$
$Sb       ab$      Error: missing b,
                   inserted (illegal A)
$S        ab$      S → AbS
$SbA      ab$      A → a
$Sba      ab$
$Sb       b$
$S        $        S → ε
$$        accept
```

For string ceadb

```
Stack        Input        Output

$S           ceadb$       S → AbS
$SbA         ceadb$       A → cAd
$SbdAc       ceadb$
$SbdA        eadb$        Error:
                          unexpected e
(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.

Error Recovery in Operator-Precedence Parsing

Error Cases:

- No relation holds between the terminal on the top of stack and the next input symbol.
- A handle is found (reduction step), but there is no production with this handle as a right side

Error Recovery:

- Each empty entry is filled with a pointer to an error routine.
- Decides the popped handle "looks like" which right hand side. And tries to recover from that situation.

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

Lecture #41

CODE OPTIMIZATION

- Code optimization is aimed at obtaining a more efficient code.
- Two constraints on the technique used to perform optimizations
  - They must ensure that the transformed program is semantically equivalent to the original program.
  - The improvement of the program efficiency must be achieved without changing the algorithms which are used in the program.
- Optimization may be classified as Machine dependent and Machine independent.
  - Machine dependent optimizations exploit characteristics of the target machine.
  - Machine independent optimizations are based on mathematical properties of a sequence of source statements.

Optimizing Transformations

Common Sub-expression Elimination

- An expression need not be evaluated if it was previously computed and values of variables in this expression have not changed since the earlier computations.

Example:

```
a = d * c;
.
.
.

d = b * c + x –y;
```

We can eliminate the second evaluation of b*c from this code if none of the intervening statements has changed its value. The code can be rewritten as given below.

```
T1 = b * c;
a = T1;
.
.
.
d = T1 + x – y;
```

Compile Time Evaluation

- We can improve the execution efficiency of a program by shifting execution time actions to compile time.
- We can evaluate an expression by a single value (known as *folding*).

Example:

```
A = 2 * (22.0/7.0) * r
```

Here we can perform the computation 2 * (22.0/7.0) at compile time itself.

• If a variable is assigned a constant value and is used in an expression without being assigned other value to it, we can evaluate some portion of the expression using the constant value (known as *Constant Propagation*).

Example:

```
x = 12.4


.
.
y = x / 2.3
```

Here we evaluate x / 2.3 as 12.4 / 2.3 at compile time.

Variable Propagation

- If a variable is assigned to another variable, we use one in place of another.
- This will be useful to carry out other optimization that were otherwise not possible.

Example:

```
c = a * b; x = a;
.
.
.
d = x * b;
```

Here, if we replace x by a then a * b and x * b will be identified as common sub- expressions.

Dead Code Elimination

•If the value contained in a variable at that point is not used anywhere in the program subsequently, the variable is said to be dead at that place.
•If an assignment is made to a dead variable, then that assignment is a dead assignment and it can be safely removed from the program.
•A piece of code is said to be dead if it computes values that are never used anywhere in the program.
• Dead Code can be eliminated safely.
• Variable propagation often leads to making assignment statement into dead code.

Example:

```
c = a * b;
x = a;
.
```

```
.
.
d = x * b + 4;
```

Variable propagation will lead to following changes. c = a * b;

```
x = a;
.
.
.
d = a * b + 4;
```

This assignment x = a is now useless and can be removed

```
c = a * b;
.
.
.
d = a * b + 4;
```

Code Motion

•We aim to improve the execution time of the program by reducing the evaluation frequency of expressions.
•Evaluation of expressions is moved from one part of the program to another in such a way that it is evaluated lesser frequently.
•Loops are usually executed several times.
•We can bring the loop-invariant statements out of the loop.

Example:

```
a = 200;
while (a > 0)
{
        b = x + y;
        if ( a%b == 0)
        printf (“%d”, a);
}
```

The statement b = x + y is executed every time with the loop. But because it is loop- invariant, we can bring it outside the loop. It will then be executed only once.

```
a = 200;
b = x + y;
while (a > 0)
        {
                if ( a%b == 0)
                printf (“%d”, a);
        }
```

Induction Variables and Strength Reduction

•An induction variable may be defined as an integer scalar variable which is used in loop for the following kind of assignments i = i + constant.
•Strength Reduction means replacing the high strength operator by a low strength operator.
•Strength Reduction used on induction variables to achieve a more efficient code.

Example:

```
i = 1;
        while (i < 10)
        {
                .
                .
                .
                y = i * 4;
                .
                .

        }
```

This code can be replaced by the following code. i = 1;

```
t = 4;
while (t < 40)
{
.
.
.
y = t;
.
.
.
t = t + 4;
.
.
.
}
```

Use of Algebraic Identities

•Certain computations that look different to the compiler and are not identified as common sub-expressions are actually same.
•An expression B op C will usually be treated as being different to C op B.
•However, for certain operations (like addition and multiplication), they will produce the same result.
•We can achieve further optimization by treating them as common sub-expressions for such operations.

Lecture #42

Local Optimizations

•Target code generated statement by statement generally contains redundant instructions.

•We can improve the quality of such code by applying optimizing transformations locally by examining a short sequence of code instructions and replacing them by faster or shorter sequence, if possible.

•This technique is known as *Peephole Optimization* where the peephole is a small moving window on the program.

• Many of the code optimization techniques can be carried out by a single portion of a program known as *Basic Block*.

Basic Block

•A basic Block is defined as a sequence of consecutive statements with only one entry (at the beginning) and one exit (at the end).

•When a Basic Block of a program is entered, all the statements are executed in sequence without a halt or possibility of branch except at the end.

•In order to determine all the Basic Block in a program, we need to identify the *leaders*, the first statement of each Basic Block.

- Any statement that satisfies the following conditions is a leader;
  - The first statement is leader.
  - Any statement which is the target of any goto (jump) is a leader.
    - Any statement that immediately follows a goto (jump) is a leader.
    - A basic block is defined as the portion of code from one leader to the statement up to but including the next leader or the end of the program.

Flow Graph

- It is a directed graph that is used to portray basic block and their successor relationships.
- The nodes of a flow graph are the basic blocks.
- The basic block whose leader is the first statement is known as the initial block.
- There is a directed edge from block B1 to B2 if B2 could immediately follow B1 during execution.
- To determine whether there should be directed edge from B1 to B2, following criteria is applied:
  - There is a jump from last statement of B1 to the first statement of B2, OR
  - B2 immediately follows B1 in order of the program and B1 does not end in an unconditional jump.
- B1 is known as the *predecessor* of B2 and B2 is a *successor* of B1.

Loops

- We need to identify all the loops in a flow graph to carry out many optimizations discussed earlier.
- A loop is a collection of nodes that
  - is strongly connected i.e. from any node in the loop to any other, there is a path of length one or more wholly within the loop, and
  - has a unique entry, a node in the loop such that the only way to reach a node in the loop from a node outside the loop is to first go through the entry.

DAG Representation of a Basic Block

- Many optimizing transformations can be implemented using the DAG representation of a basic block.
- DAG stands for Directed Acyclic Graph i.e. a graph with directed edges and no cycles.
- DAG is very much like a tree but differs in that it may contain shared nodes where shared nodes indicate common sub-expressions.
- A DAG has following components;
  - Leaves are labeled by unique identifiers, either variable names or constants.
  - Interior nodes are labeled by an operator symbol.
  - Nodes are optionally given an extra set of identifiers known as *attached identifiers*.

DAG Construction

- We assume there are initially no nodes and NODE ( ) is undefined for all arguments.
- The 3-address statements has one of three cases:

  (i) A = B op C

  (ii) A = op B

  (iii) A = B
- We shall do the following steps (1) through (3) for each 3-address statement of the basic block:

(1) If NODE (B) is undefined, create a leaf labeled B, and let NODE (B) be this node. In case (i), if NODE (C) is undefined, create a leaf labeled C and let that leaf be NODE (C); (2) In case (i), determine if there is a node labeled op whose left child is NODE (B) and whose right child is NODE (C). (This is to catch common sub-expressions.) If not create such a node. In case (ii), determine whether there is a node labeled op whose lone child is NODE (B). If not create such a node. Let n be the node found or created in both cases. In case (iii), let n be NODE (B).

(3) Append A to the list of attached identifiers for the node n in (2). Delete A from the list of attached identifiers for NODE (A). Finally, set NODE (A) to n.

Applications of DAG

- We automatically detect common sub-expressions while constructing DAG.
- It is also known as to which identifiers have there values used in side the block; they are exactly those for which a leaf is created in Step (1).
- Wecan also determine which statements compute values which could be used outside the block; they are exactly those statements S whose node n in step (2) still has NODE (A) = n at the end of DAG construction, where A is the identifier assigned by statement S i. e. A is still an attached identifier for n.

Global Data Flow Analysis

- Certain optimizations can be achieved by examining the entire program and not just a portion of the program.
- User-defined chaining is one particular problem of this kind.
- Here we try to find out as to which definition of a variable is applicable in a statement using the value of that variable.