



NPTEL ONLINE CERTIFICATION COURSES

Compiler Design

Introduction

Santanu Chattopadhyay
Electronics and Electrical Communication Engineering

CONCEPTS COVERED

- What is a Compiler
- Compiler Applications
- Phases of a Compiler
- Challenges in Compiler Design
- Compilation Process – An Example
- Conclusion



Introduction

- Compilers have become part and parcel of computer systems
- Makes user's computing requirements, specified as a piece of program, understandable to the underlying machine
- Complex transformation
- Large number of options in terms of advances in computer architecture, memory management and newer operating systems

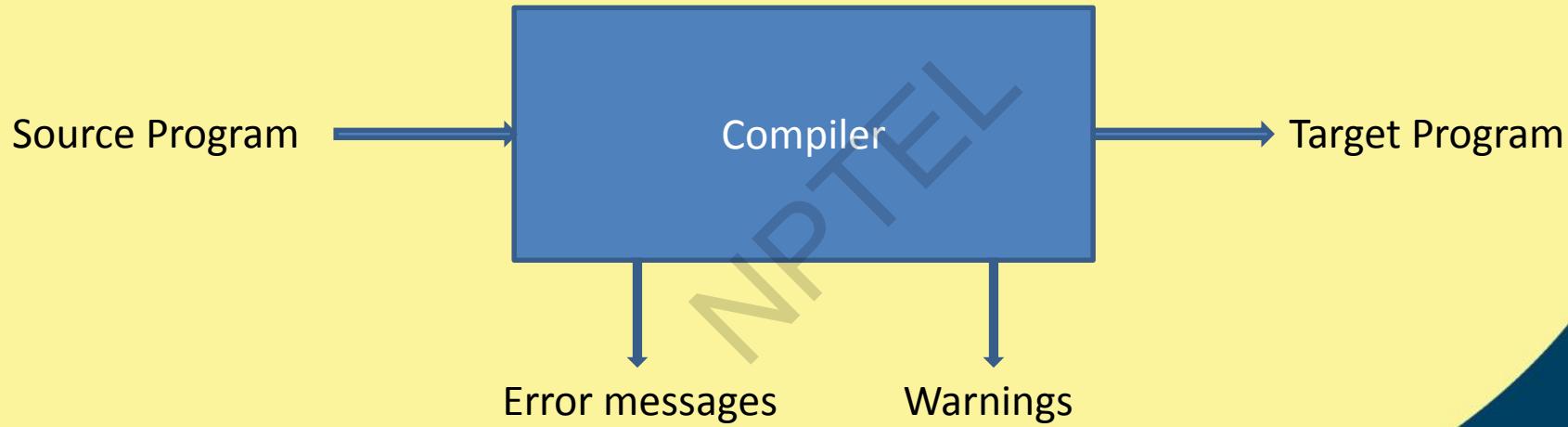


What is a compiler

- A system software to convert source language program to target language program
- Validates input program to the source language specification – produces error messages / warnings
- Primitive systems did not have compilers, programs written assembly language, handcoded into machine code
- Compiler design started with FORTRAN in 1950s
- Many tools have been developed for compiler design automation



Compiler Input-Output



How Many Compilers ??

- Impossible to quantify number of compilers designed so far
- Large number of well known, lesser known and possibly unknown computer languages designed so far
- Equally large number of hardware-software platforms
- Efficient compilers must for survival of programming languages
- Inefficient translators developed for LISP made programs run very slowly
- Advances in memory management policies, particularly garbage collection, has paved the way for faster implementation of such translators, rejuvenating such languages



Compiler Applications

- Machine Code Generation
 - Convert source language program to machine understandable one
 - Takes care of semantics of varied constructs of source language
 - Considers limitations and specific features of target machine
 - Automata theory helps in syntactic checks – valid and invalid programs
 - Compilation also generate code for syntactically correct programs



Compiler Applications (Contd.)

- Format Converters
 - Act as interfaces between two or more software packages
 - Compatibility of input-output formats between tools coming from different vendors
 - Also used to convert heavily used programs written in some older languages (like COBOL) to newer languages (like C/C++)



Compiler Applications (Contd.)

- Silicon Compilation
 - Automatically synthesize a circuit from its behavioural description in languages like VHDL, Verilog etc.
 - Complexity of circuits increasing with reduced time-to-market
 - Optimization criteria for silicon compilation are area, power, delay, etc.



Compiler Applications (Contd.)

- Query Optimization
 - In the domain of database query processing
 - Optimize search time
 - More than one evaluation sequence for each query
 - Cost depends upon relative sizes of tables, availability of indexes
 - Generate proper sequence of operations suitable for fastest query processing



Compiler Applications (Contd.)

- Text Formatting
 - Accepts an ordinary text file as input having formatting commands embedded
 - Generates formatted text
 - Example *troff*, *nroff*, *LaTex* etc.



Phases of a Compiler

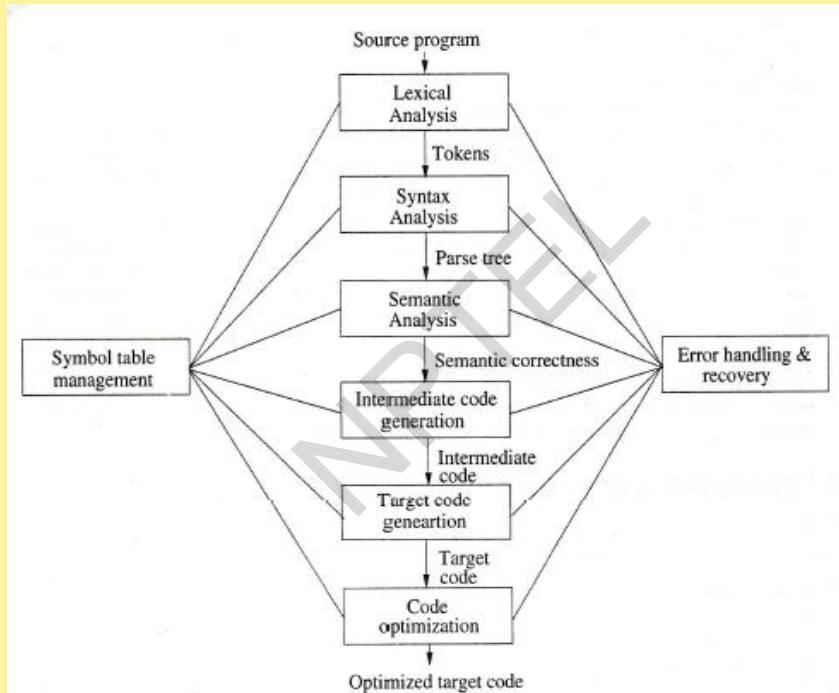
- Conceptually divided into a number of phases
 - Lexical Analysis
 - Syntax Analysis
 - Semantic Analysis
 - Intermediate Code Generation
 - Target Code Generation
 - Code Optimization
 - Symbol Table Management
 - Error Handling and Recovery

Does not exist any hard demarcation between the modules.

Work in hand-in-hand interspersed manner



Phases of a Compiler



Lexical Analysis

- Interface of the compiler to the outside world
- Scans input source program, identifies valid words of the language in it
- Also removes cosmetics, like extra white spaces, comments etc. from the program
- Expands user-defined macros
- Reports presence of foreign words
- May perform case-conversion
- Generates a sequence of integers, called *tokens* to be passed to the syntax analysis phase – later phases need not worry about program text
- Generally implemented as a *finite automata*



Syntax Analysis

- Takes words/tokens from lexical analyzer
- Works hand-in-hand with lexical analyzer
- Checks syntactic (grammatical) correctness
- Identifies sequence of grammar rules to derive the input program from the start symbol
- A *parse tree* is constructed
- Error messages are flashed for syntactically incorrect program



Semantic Analysis

- Semantics of a program is dependent on the language
- A common check is for types of variables and expressions
- Applicability of operators to operands
- *Scope rules* of the language are applied to determine types – *static scope* and *dynamic scope*

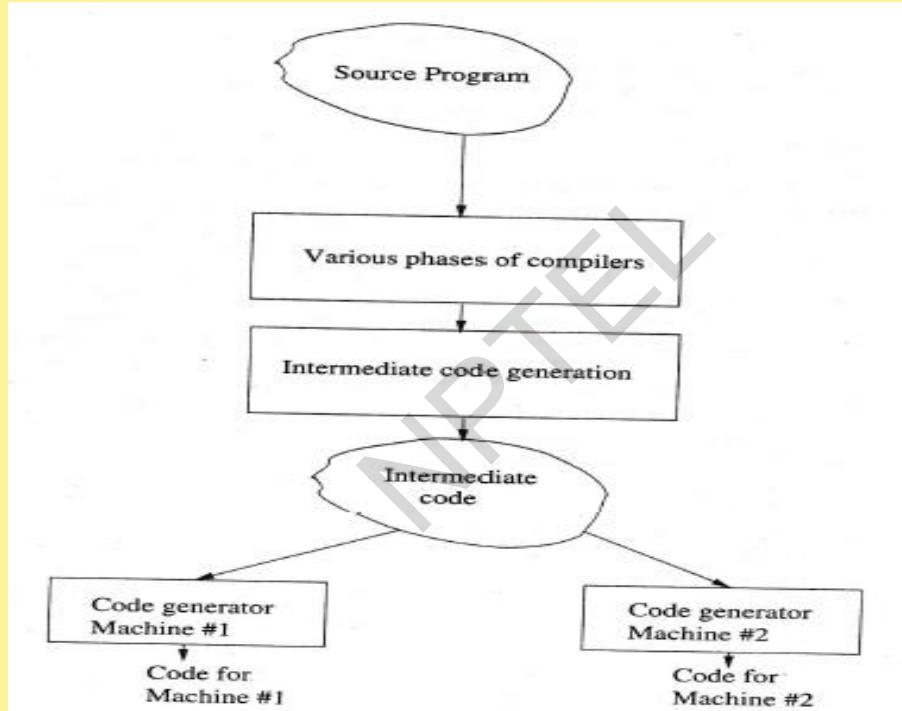


Intermediate Code Generation

- Optional towards target code generation
- Code corresponding to input source language program is generated in terms of some hypothetical machine instructions
- Helps to retarget the code from one processor to another
- Simple language supported by most of the contemporary processors
- Powerful enough to express programming language constructs



Targeting Different Machines



Target Code Generation

- Uses template substitution from intermediate code
- Predefined target language templates are used to generate final code
- Machine instructions, addressing modes, CPU registers play vital roles
- Temporary variables (user defined and compiler generated) are packed into CPU registers



Code Optimization

- Most vital step in target code generation
- Automated steps of compilers generate lot of redundant codes that can possibly be eliminated
- Code is divided into *basic blocks* – a sequence of statements with single entry and single exit
- *Local optimizations* restrict within a single basic block
- *Global optimization* spans across the boundaries of basic blocks
- Most important source of optimization are the *loops*
- Algebraic simplifications, elimination of load-and-store are common optimizations



Symbol Table Management

- Symbol table is a data structure holding information about all symbols defined in the source program
- Not part of the final code, however used as reference by all phases of a compiler
- Typical information stored there include name, type, size, relative offset of variables
- Generally created by lexical analyzer and syntax analyzer
- Good data structures needed to minimize searching time
- The data structure may be flat or hierarchical



Error Handling and Recovery

- An important criteria for judging quality of compiler
- For a semantic error, compiler can proceed
- For syntax error, parser enters into a erroneous state
- Needs to undo some processing already carried out by the parser for recovery
- A few more tokens may need to be discarded to reach a descent state from which the parser may proceed
- Recovery is essential to provide a bunch of errors to the user, so that all of them may be corrected together instead of one-by-one



Challenges in Compiler Design

- Language semantics
- Hardware platform
- Operating system and system software
- Error handling
- Aid in debugging
- Optimization
- Runtime environment
- Speed of compilation



Language Semantics

- “case” – fall through or not
- “loop” – index may or may not remember last value
- “break” and “next” modify execution sequence of the program



Hardware Platform

- Code generation strategy for accumulator based machine cannot be similar to a stack based machine
- CISC vs. RISC instructions sets



Operating System and System Software

- Format of file to be executed is depicted by the OS (more specifically, *loader*)
- Linking process can combine object files generated by different compilers into one executable file



Error Handling

- Show appropriate error messages – detailed enough to pinpoint the error, not too verbose to confuse
- A missing semicolon may be reported as “<line no> ; expected” rather than “syntax error”
- If a variable is reported to be undefined at one place, should not be reported again and again
- Compiler designer has to imagine the probable types of mistakes, design suitable detection and recovery mechanism
- Some compilers even go to the extent of modifying source program partially, in order to correct it



Aid in Debugging

- Debugging helps in detecting logical mistakes in a program
- User needs to control execution of machine language program from the source language level
- Compiler has to generate extra information regarding the correspondence between source and machine instructions
- Symbol table also needs to be available to the debugger
- Extra debugging information embedded into the machine code



Optimization

- Needs to identify set of transformations that will be beneficial for most of the programs in a language
- Transformations should be safe
- Trade-off between the time spent to optimize a program vis-a-vis improvement in execution time
- Several levels of optimizations are often used
- Selecting a debugging option may disable any optimization that disturbs the correspondence between the source program and object code



Runtime Environment

- Deals with creating space for parameters and local variables
- For languages like FORTRAN, it is static – fixed memory locations created for them
- Not suitable for languages supporting recursion
- To support recursion, stack frames are used to hold variables and parameters



Speed of Compilation

- An important criteria to judge the acceptability of a compiler to the user community
- Initial phase of program development contains lots of bugs, hence quick compilation may be the objective, rather than optimized code
- Towards the final stages, execution efficiency becomes the prime concern, more compilation time may be afforded to optimize the machine code significantly



Compilation – An Example

Consider the following program:

program

 var X1, X2: integer; {integer variable declaration}

 var XR1: real; {real variable declaration}

begin

 XR1 := X1 + X2 * 10; {assignment statement}

end.



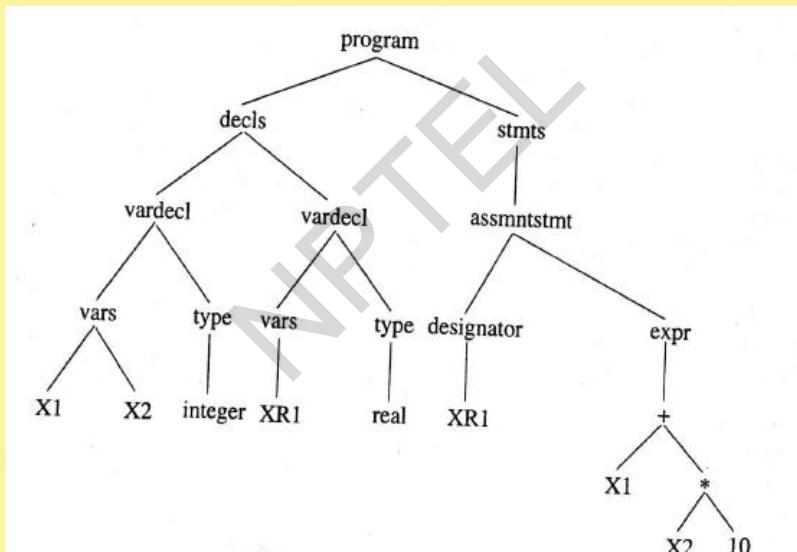
Lexical Analysis

- Produces the following sequence of tokens:
program, var, X1, ',', X2, ':', integer, ':', var, XR1, ':', real, ';', begin, XR1,
':=', X1, '+', X2, '*', 10, ';', end, '.'
- Whitespaces and comments are discarded and removed by the Lexical Analyzer



Syntax Analysis

- Assuming the program to be grammatically correct, the following parse tree is produced



Code Generation

- Assuming a stack oriented machine with instructions like PUSH, ADD, MULT, STORE, the code looks like

PUSH X2

PUSH 10

MULT

PUSH X1

ADD

PUSH @XR1 @ symbol returns the address of a variable

STORE



Symbol Table

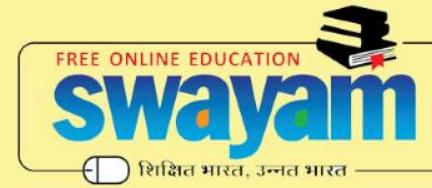
Name	Class	Type
X1	Variable	Integer
X2	Variable	Integer
XR1	Variable	Real



Conclusion

- Seen an overview of the compiler design process
- Different phases of a compiler have been enumerated
- Challenges faced by the compiler designer have been noted
- An example compilation process has been illustrated





NPTEL ONLINE CERTIFICATION COURSES

Thank
you!



NPTEL ONLINE CERTIFICATION COURSES

Compiler Design

Lexical Analysis

Santanu Chattpadhyay

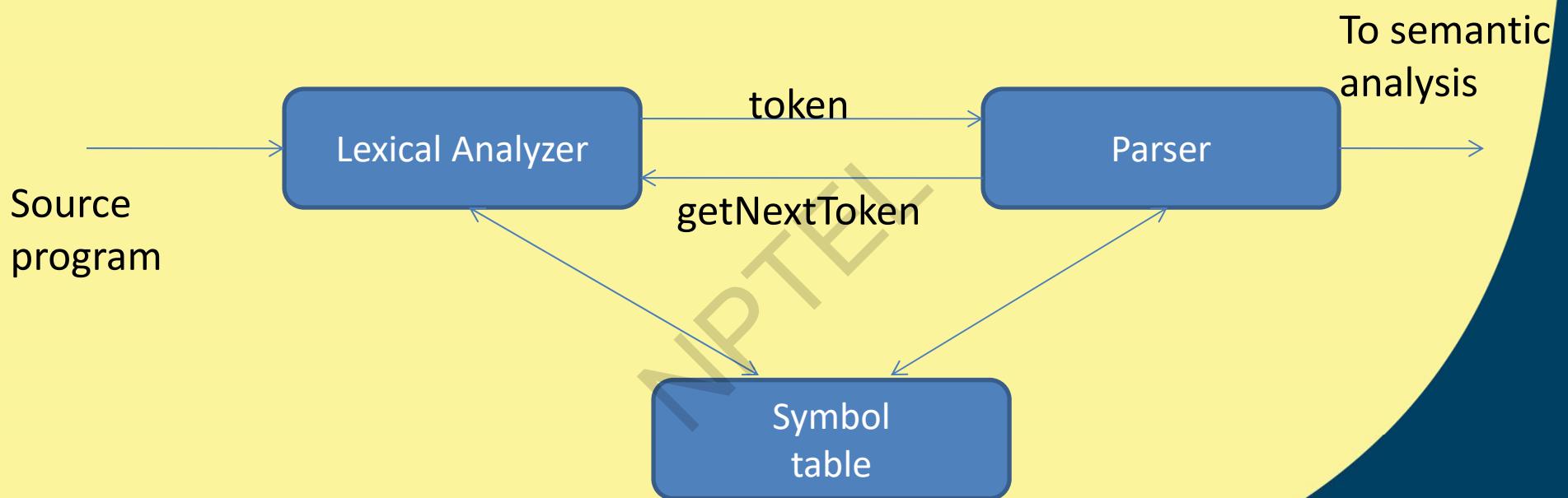
Electronics and Electrical Communication Engineering

CONCEPTS COVERED

- Role of Lexical Analyzer
- Tokens, Patterns, Lexemes
- Lexical Errors and Recovery
- Specification of Tokens
- Recognition of Tokens
- Finite Automata
- NFA and DFA
- Tool lex
- Conclusion



Role of lexical analyzer



Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability



Tokens, Patterns and Lexemes

- A token is a pair – a token name and an optional token value
- A pattern is a description of the form that the lexemes of a token may take
- A lexeme is a sequence of characters in the source program that matches the pattern for a token



Example

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 letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but “ surrounded by “	“core dumped”



Attributes for tokens

- $E = M * C ^\star 2$
 - <id, pointer to symbol table entry for E>
 - <assign-op>
 - <id, pointer to symbol table entry for M>
 - <mult-op>
 - <id, pointer to symbol table entry for C>
 - <exp-op>
 - <number, integer value 2>



Lexical errors

- Some errors are out of power of lexical analyzer to recognize:
 - $fi (a == f(x)) \dots$
- However it may be able to recognize errors like:
 - $d = 2r$
- Such errors are recognized when no pattern for tokens matches a character sequence



Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters



Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
 - In C language: we need to look after -, = or < to decide what token to return
 - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely

E = M * C ** 2 eof



Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
 - $\text{letter}(\text{letter} \mid \text{digit})^*$
- Each regular expression is a pattern specifying the form of strings



Regular Expressions

- ϵ is a regular expression denoting the language $L(\epsilon) = \{\epsilon\}$, containing only the empty string
- If a is a symbol in Σ then a is a regular expression, $L(a) = \{a\}$
- If r and s are two regular expressions with languages $L(r)$ and $L(s)$, then
 - $r|s$ is a regular expression denoting the language $L(r) \cup L(s)$, containing all strings of $L(r)$ and $L(s)$
 - rs is a regular expression denoting the language $L(r)L(s)$, created by concatenating the strings of $L(s)$ to $L(r)$
 - r^* is a regular expression denoting $(L(r))^*$, the set containing zero or more occurrences of the strings of $L(r)$
 - (r) is a regular expression corresponding to the language $L(r)$



Regular definitions

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

...

$d_n \rightarrow r_n$

- Example:

letter_ $\rightarrow A | B | \dots | Z | a | b | \dots | z | _$

digit $\rightarrow 0 | 1 | \dots | 9$

id $\rightarrow \text{letter}_\text{ } (\text{letter}_\text{ } | \text{digit})^*$



Extensions

- One or more instances: $(r)^+$
- Zero or one instances: $r^?$
- Character classes: [abc]
- Example:
 - letter_ -> [A-Za-z_]
 - digit -> [0-9]
 - id -> letter_(letter_|digit)*



Examples with $\Sigma = \{0, 1\}$

- $(0|1)^*$: All binary strings including the empty string
- $(0|1)(0|1)^*$: All nonempty binary strings
- $0(0|1)^*0$: All binary strings of length at least 2, starting and ending with 0s
- $(0|1)^*0(0|1)(0|1)(0|1)$: All binary strings with at least three characters in which the third-last character is always 0
- $0^*10^*10^*10^*$: All binary strings possessing exactly three 1s



Example

Set of floating-point numbers:

$$(+ | - | \varepsilon) \text{ digit } (\text{digit})^* (.\text{ digit } (\text{digit})^* | \varepsilon) ((E (+ | - | \varepsilon) \text{ digit } (\text{digit})^*) | \varepsilon)$$


Recognition of tokens

- Starting point is the language grammar to understand the tokens:

stmt -> if expr **then** stmt

| if expr **then** stmt **else** stmt

| ϵ

expr -> term **relop** term

| term

term -> **id**

| **number**



Recognition of tokens (cont.)

- The next step is to formalize the patterns:

digit → [0-9]

Digits → digit+

number → digit(.digits)? (E[+−]? Digit)?

letter → [A-Za-z_]

id → letter (letter|digit)*

If → if

Then → then

Else → else

Relop → < | > | <= | >= | = | <>

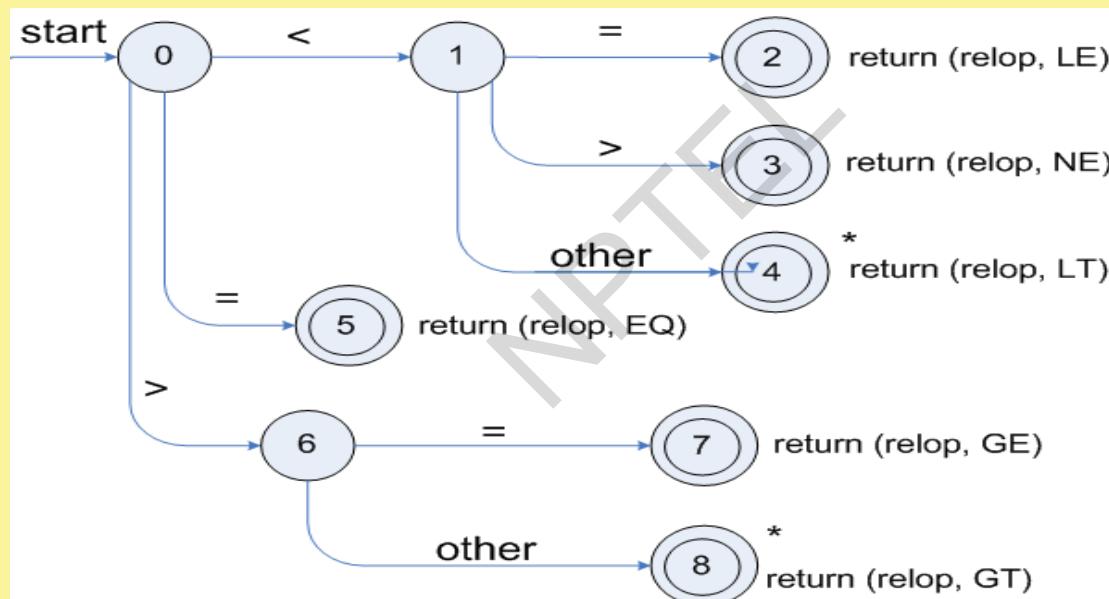
- We also need to handle whitespaces:

ws → (blank | tab | newline)+



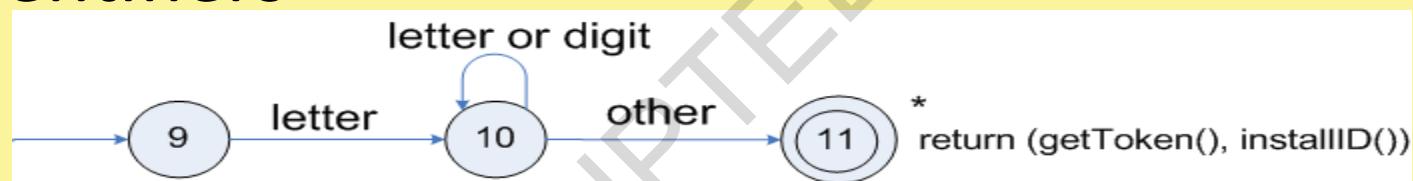
Transition diagrams

- Transition diagram for relop



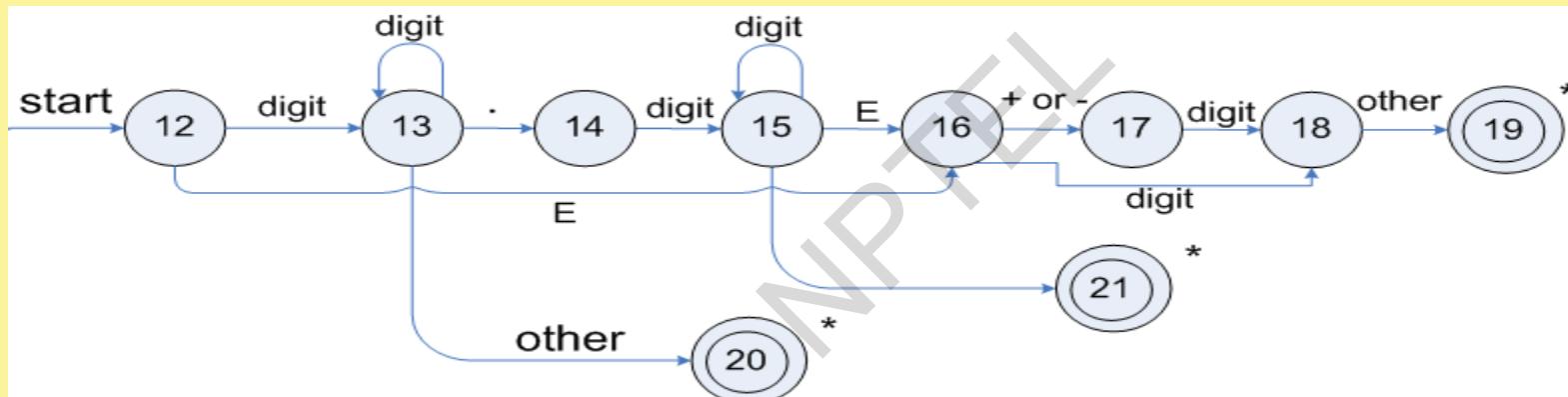
Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers



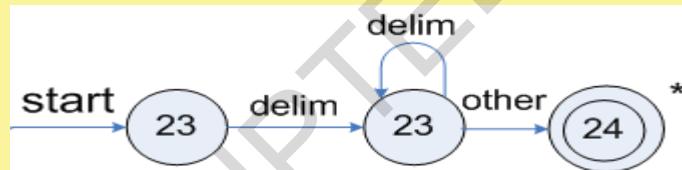
Transition diagrams (cont.)

- Transition diagram for unsigned numbers



Transition diagrams (cont.)

- Transition diagram for whitespace



Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP)
    while (1) { /* repeat character processing until a
                    return or failure occurs */
        switch(state) {
            case 0: c= nextchar();
                if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else fail(); /* lexeme is not a relop */
                break;
            case 1: ...
            ...
            case 8: retract();
                retToken.attribute = GT;
                return(retToken);
        }
    }
}
```



Finite Automata

- Regular expressions = specification
- Finite automata = implementation
- A finite automaton consists of
 - An input alphabet Σ
 - A set of states S
 - A start state n
 - A set of accepting states $F \subseteq S$
 - A set of transitions $\text{state} \xrightarrow{\text{input}} \text{state}$



Finite Automata

- Transition

$$s_1 \xrightarrow{a} s_2$$

- Is read

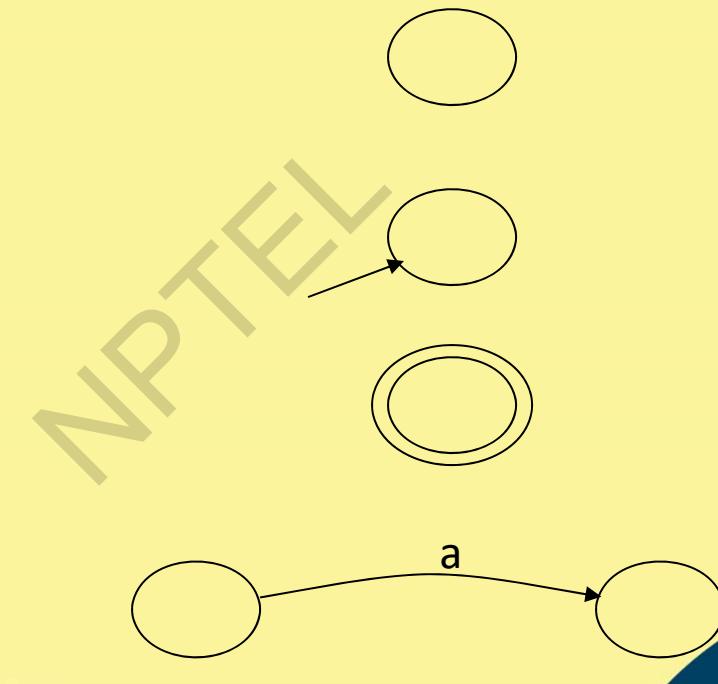
In state s_1 on input “a” go to state s_2

- If end of input
 - If in accepting state => accept, otherwise => reject
- If no transition possible => reject



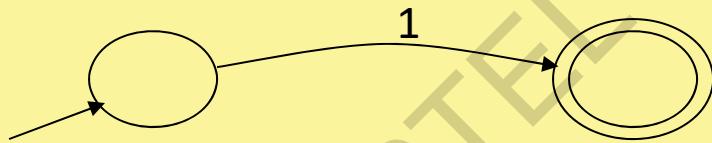
Finite Automata State Graphs

- A state
- The start state
- An accepting state
- A transition



A Simple Example

- A finite automaton that accepts only “1”

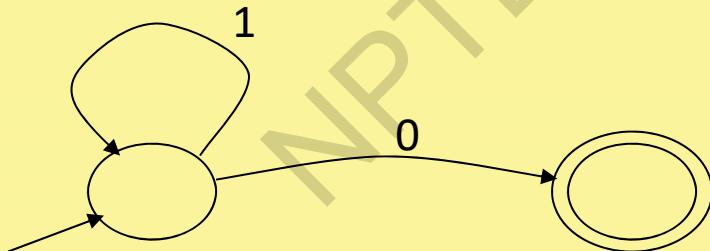


- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state



Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}

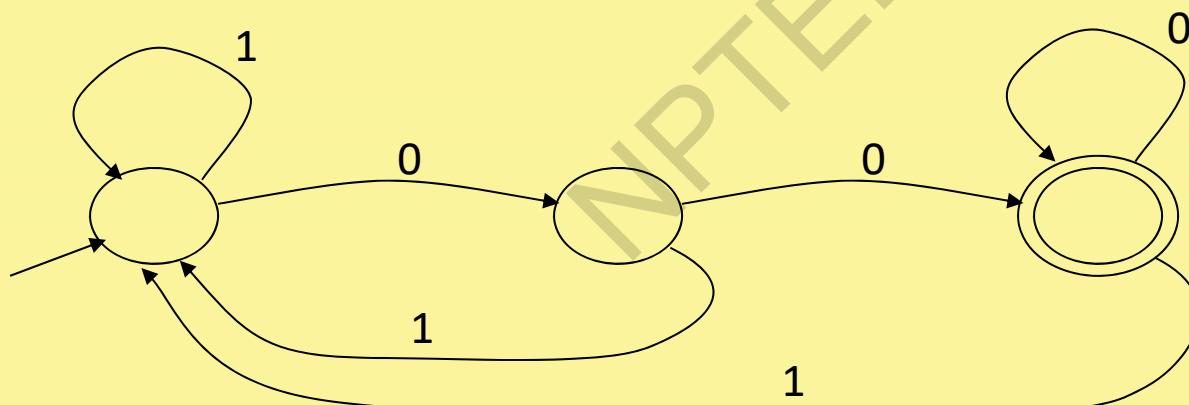


- Check that “1110” is accepted but “110...” is not



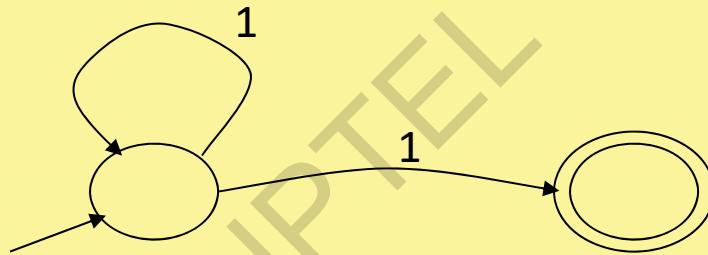
And Another Example

- Alphabet {0,1}
- What language does this recognize?



And Another Example

- Alphabet still { 0, 1 }

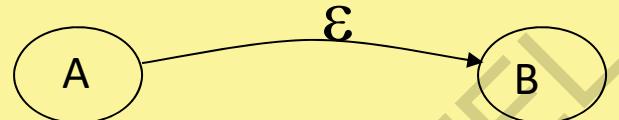


- The operation of the automaton is not completely defined by the input
 - On input “11” the automaton could be in either state



Epsilon Moves

- Another kind of transition: ϵ -moves



- Machine can move from state A to state B without reading input



Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
 - One transition per input per state
 - No ϵ -moves
- Nondeterministic Finite Automata (NFA)
 - Can have multiple transitions for one input in a given state
 - Can have ϵ -moves
- *Finite* automata have *finite* memory
 - Need only to encode the current state



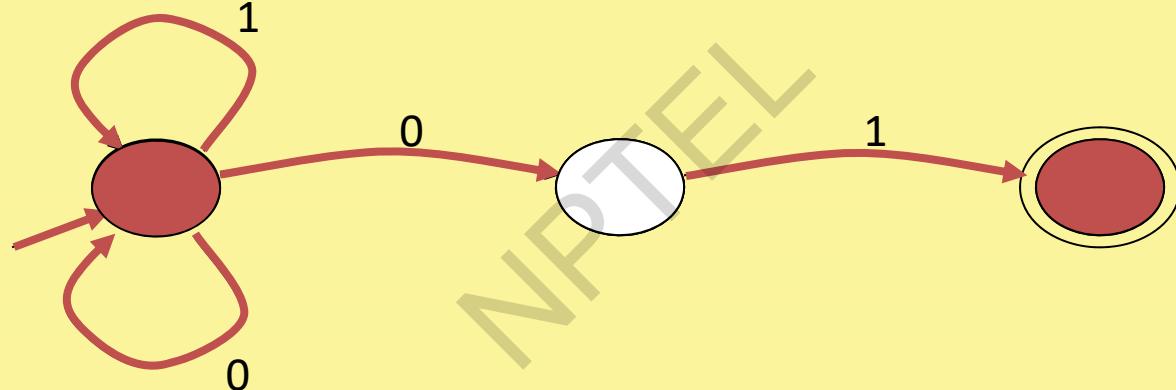
Execution of Finite Automata

- A DFA can take only one path through the state graph
 - Completely determined by input
- NFAs can choose
 - Whether to make ϵ -moves
 - Which of multiple transitions for a single input to take



Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts if it can get in a final state



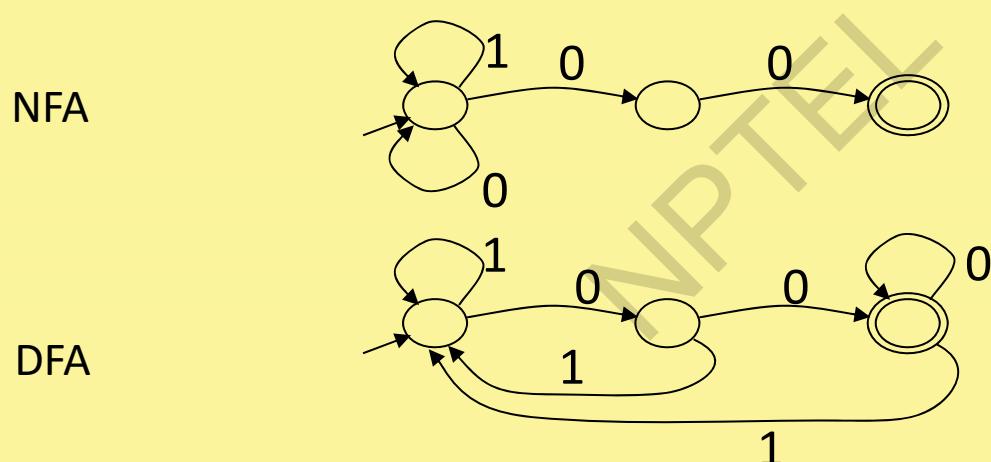
NFA vs. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are easier to implement
 - There are no choices to consider



NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

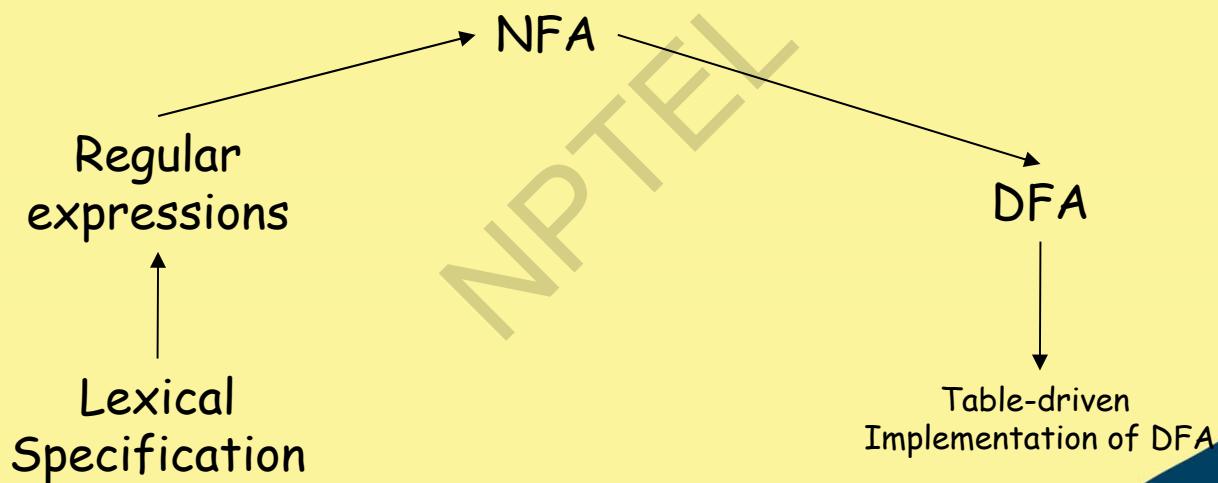


- DFA can be exponentially larger than NFA



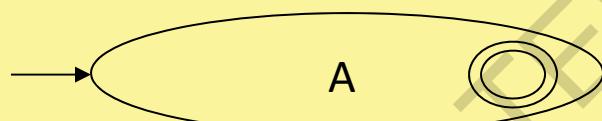
Regular Expressions to Finite Automata

- High-level sketch



Regular Expressions to NFA (1)

- For each kind of rexp, define an NFA
 - Notation: NFA for rexp A



- For ϵ

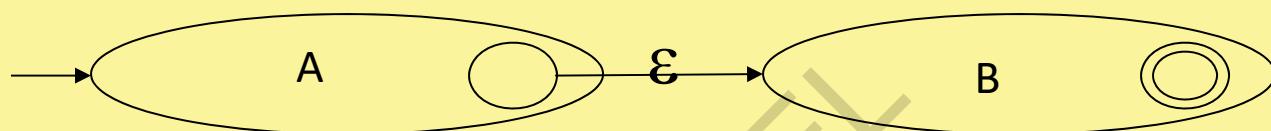


- For input a

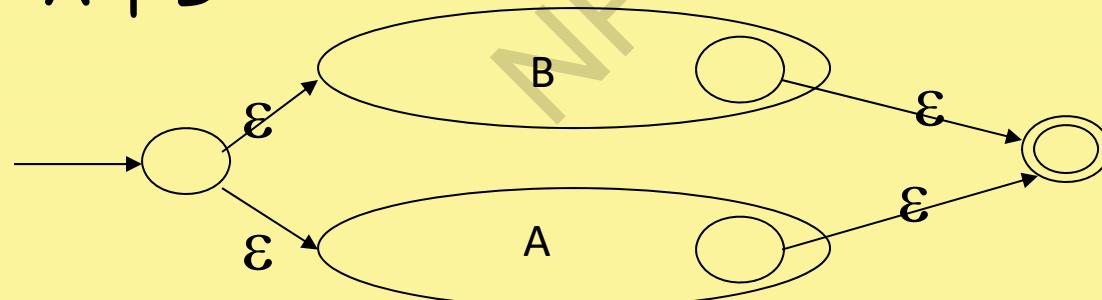


Regular Expressions to NFA (2)

- For AB

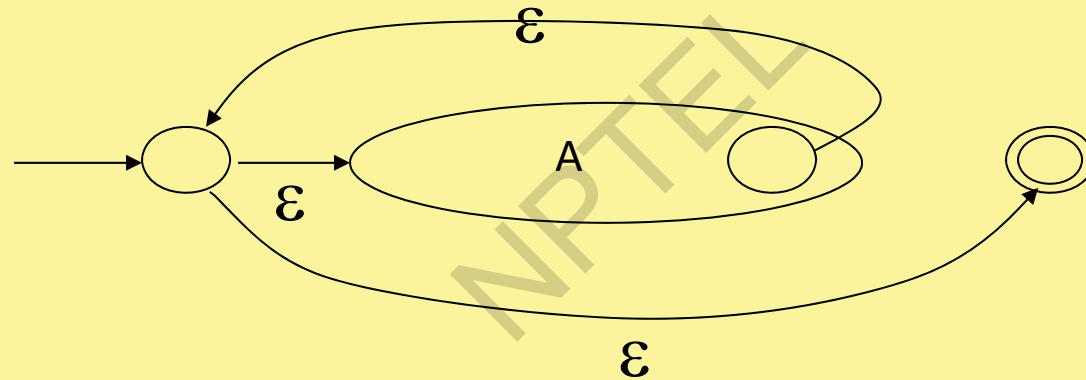


- For $A \mid B$



Regular Expressions to NFA (3)

- For A^*

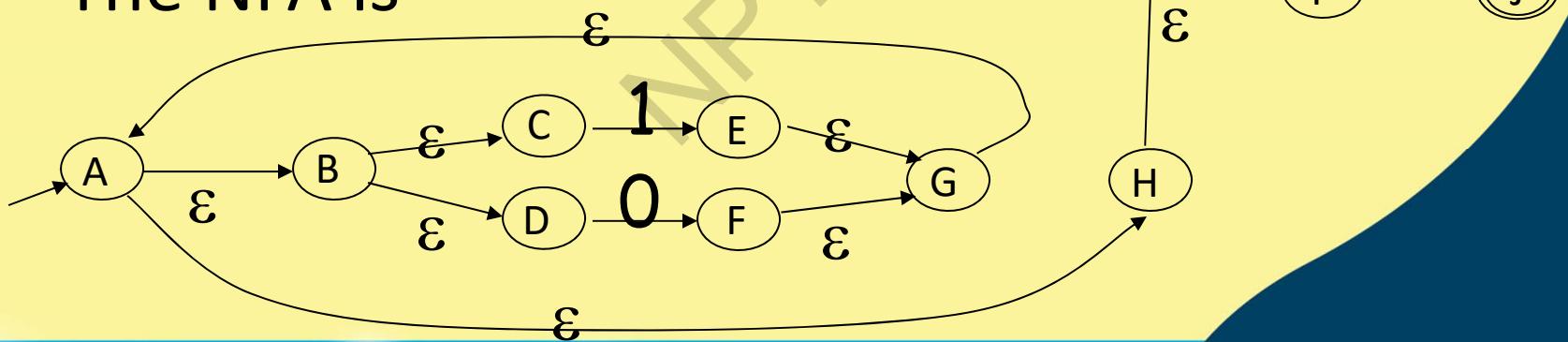


Example of RegExp -> NFA conversion

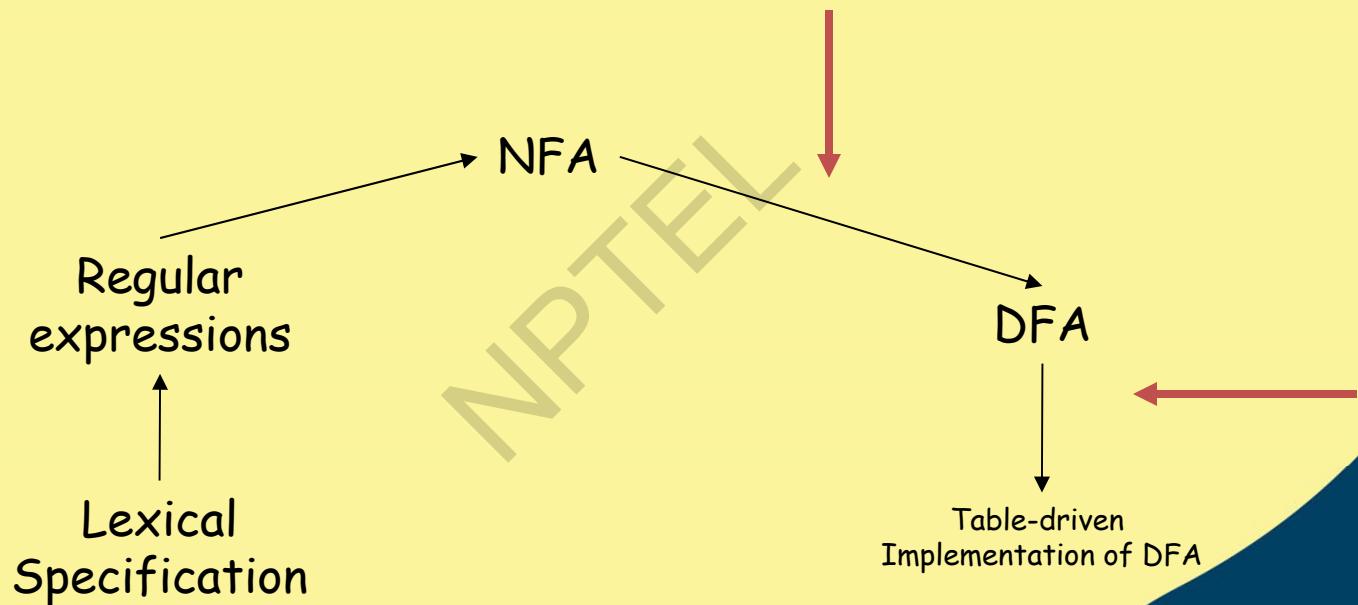
- Consider the regular expression

$$(1 \mid 0)^*1$$

- The NFA is



Next

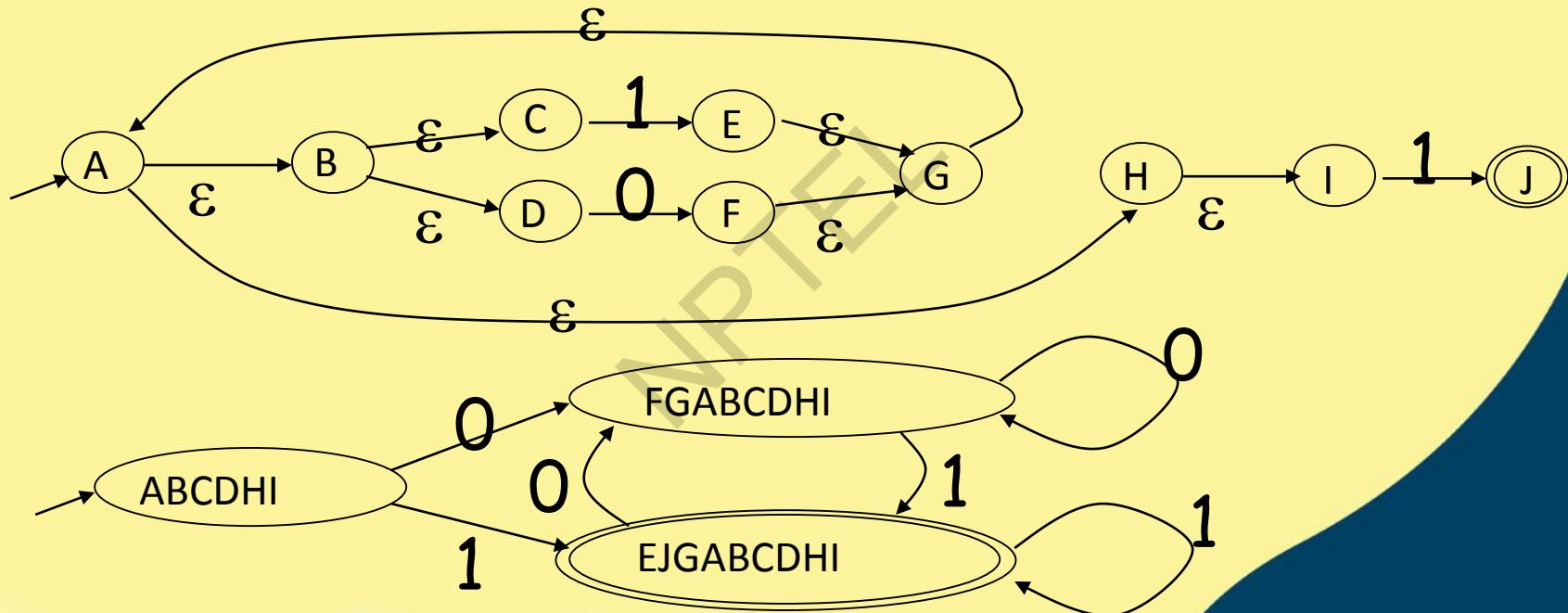


NFA to DFA. The Trick

- Simulate the NFA
- Each state of resulting DFA
 - = a non-empty subset of states of the NFA
- Start state
 - = the set of NFA states reachable through ϵ -moves from NFA start state
- Add a transition $S \rightarrow^a S'$ to DFA iff
 - S' is the set of NFA states reachable from the states in S after seeing the input a
 - considering ϵ -moves as well



NFA \rightarrow DFA Example



NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?
 - $2^N - 1$ = finitely many, but exponentially many

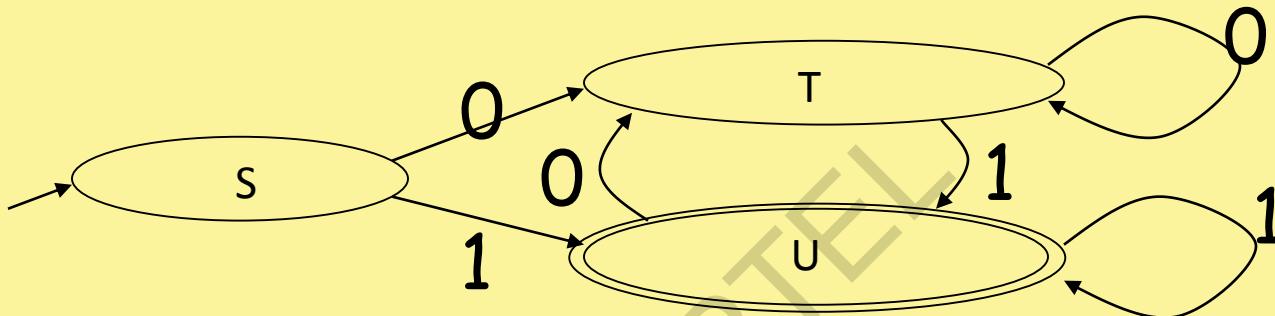


Implementation

- A DFA can be implemented by a 2D table T
 - One dimension is “states”
 - Other dimension is “input symbols”
 - For every transition $S_i \xrightarrow{a} S_k$ define $T[i,a] = k$
- DFA “execution”
 - If in state S_i and input a , read $T[i,a] = k$ and skip to state S_k
 - Very efficient



Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

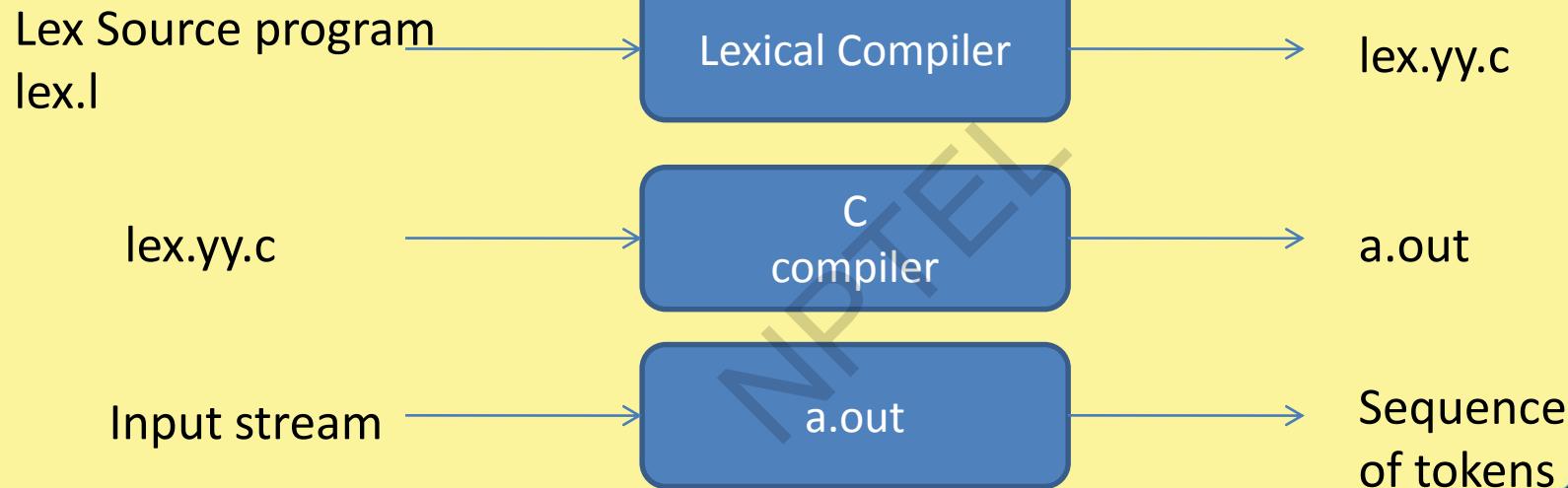


Implementation (Cont.)

- NFA → DFA conversion is at the heart of tools such as lex, flex or jflex
- But, DFAs can be huge
- In practice, lex-like tools trade off speed for space in the choice of NFA and DFA representations



Lexical Analyzer Generator - Lex



Structure of Lex programs

declarations

%%

translation rules

%%

auxiliary functions



Pattern {Action}



Example

```
%{  
    /* definitions of manifest constants  
    LT, LE, EQ, NE, GT, GE,  
    IF, THEN, ELSE, ID, NUMBER, RELOP */  
}  
  
/* regular definitions  
delim      [ \t\n]  
ws         {delim}+  
letter[A-Za-z]  
digit [0-9]  
id          {letter}({letter})|{digit})*  
number      {digit}+|(\.{digit}+)?(E[+-]?{digit}+)?  
  
%  
{ws} /* no action and no return */  
if          {return(IF);}  
then        {return(THEN);}  
else        {return(ELSE);}  
{id}        {yyval = (int) installID(); return(ID); }  
{number}    {yyval = (int) installNum(); return(NUMBER);} 
```

```
int installID() /* funtion to install the  
lexeme, whose first character is  
pointed to by yytext, and whose  
length is yyleng, into the symbol  
table and return a pointer thereto  
*/  
} 
```

```
int installNum() /* similar to  
installID, but puts numerical  
constants into a separate table */  
} 
```



Conclusion

- Words of a language can be specified using regular expressions
- NFA and DFA can act as acceptors
- Regular expressions can be converted to NFA
- NFA can be converted to DFA
- Automated tool lex can be used to generate lexical analyser for a language





NPTEL ONLINE CERTIFICATION COURSES

Thank
you!

Compiler Design

Syntax Analysis

Santanu Chattopadhyay

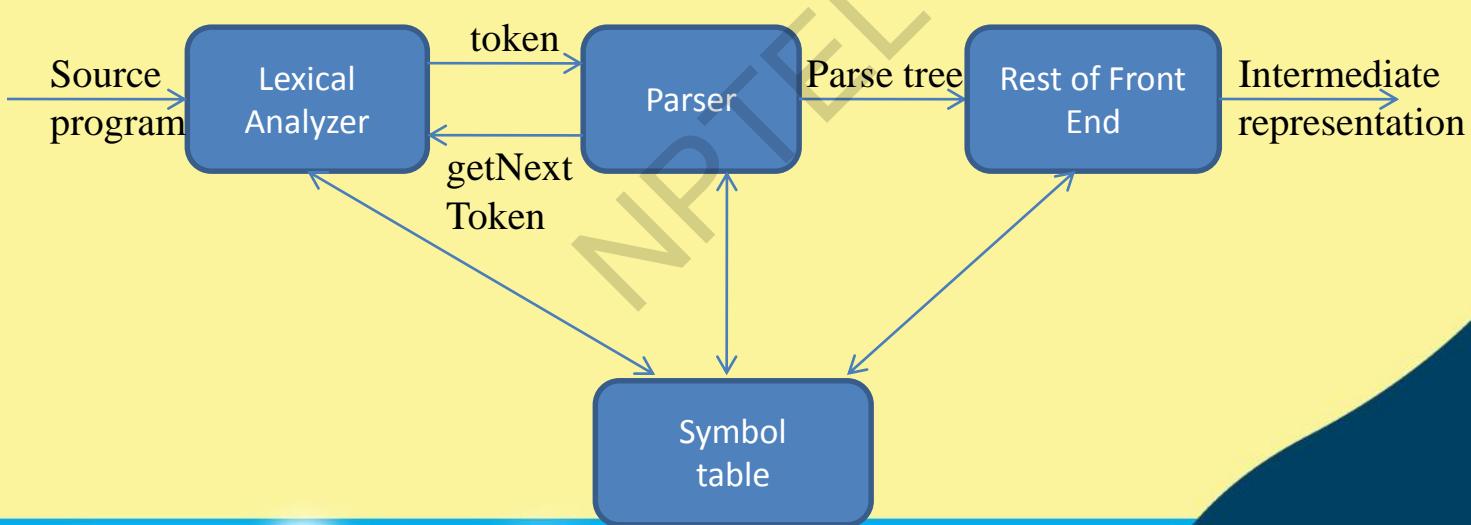
Electronics and Electrical Communication Engineering



- Role of Parsers
- Context Free Grammars
- Top-Down Parsing
- Bottom-Up Parsing
- Parser Generators
- Conclusion



The role of parser



Grammar

- A 4-tuple $G = \langle V_N, V_T, P, S \rangle$ of a language $L(G)$
 - V_N is a set of nonterminal symbols used to write the grammar
 - V_T is the set of terminals (set of words in the language $L(G)$)
 - P is a set of production rules
 - S is a special symbol in V_N , called the start symbol of the grammar
- Strings in language $L(G)$ are those derived from S by applying the production rules from P
- Examples:

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


Grammar

- A 4-tuple $G = \langle V_N, V_T, P, S \rangle$ of a language $L(G)$
 - V_N is a set of nonterminal symbols used to write the grammar
 - V_T is the set of terminals (set of words in the language $L(G)$)
 - P is a set of production rules
 - S is a special symbol in V_N , called the start symbol of the grammar
- Strings in language $L(G)$ are those derived from S by applying the production rules from P
- Examples:

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


Error handling

- Common programming errors
 - Lexical errors
 - Syntactic errors
 - Semantic errors
 - Lexical errors
- Error handler goals
 - Report the presence of errors clearly and accurately
 - Recover from each error quickly enough to detect subsequent errors
 - Add minimal overhead to the processing of correct programs



Error-recovery strategies

- Panic mode recovery
 - Discard input symbol one at a time until one of designated set of synchronization tokens is found
- Phrase level recovery
 - Replacing a prefix of remaining input by some string that allows the parser to continue
- Error productions
 - Augment the grammar with productions that generate the erroneous constructs
- Global correction
 - Choosing minimal sequence of changes to obtain a globally least-cost correction



Context free grammars

- Terminals
- Nonterminals
- Start symbol
- Productions

expression -> expression + term
expression -> expression – term
expression -> term
term -> term * factor
term -> term / factor
term -> factor
factor -> (expression)
factor -> **id**

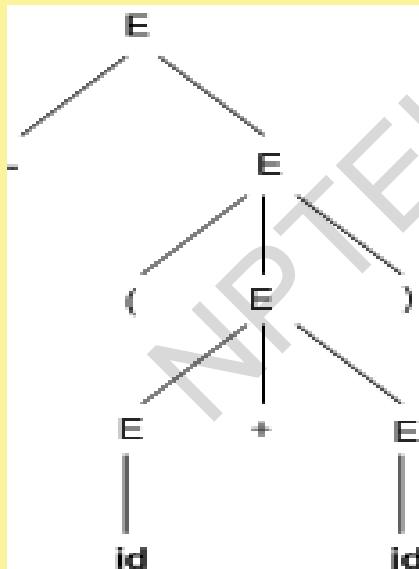


Derivations

- Productions are treated as rewriting rules to generate a string
- Rightmost and leftmost derivations
 - $E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$
 - Derivations for $-(id+id)$
 - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$
 - $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

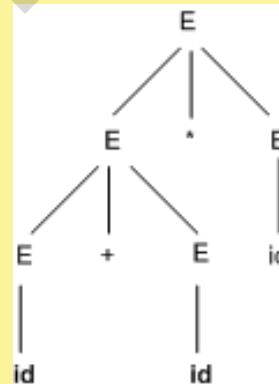
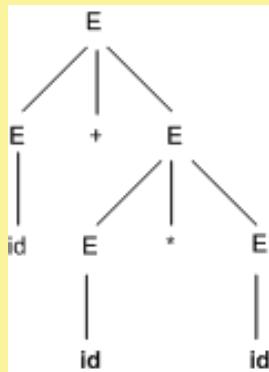


Parse tree



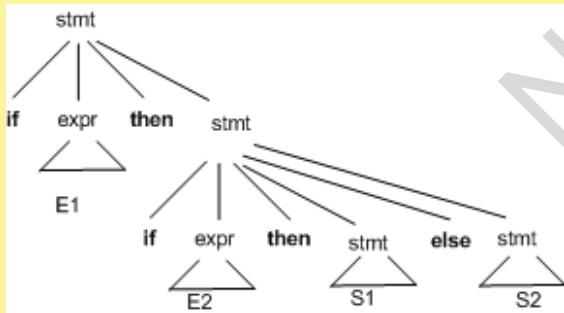
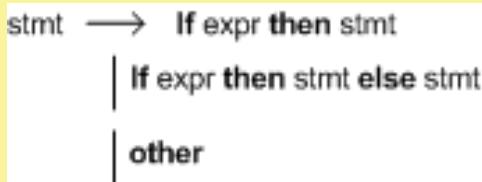
Ambiguity

- For some strings there exist more than one parse tree
- Or more than one leftmost derivation
- Or more than one rightmost derivation
- Example: $\text{id} + \text{id} * \text{id}$

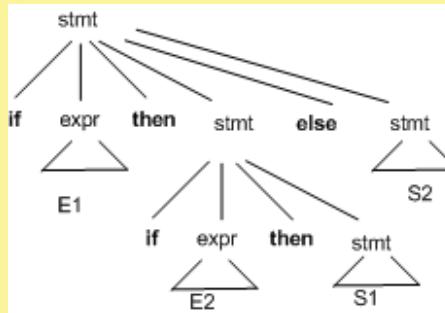
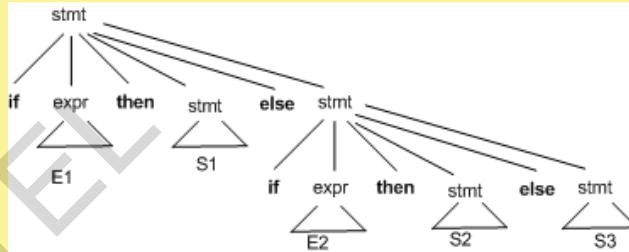


Elimination of ambiguity

if E1 then if E2 then S1 else S2

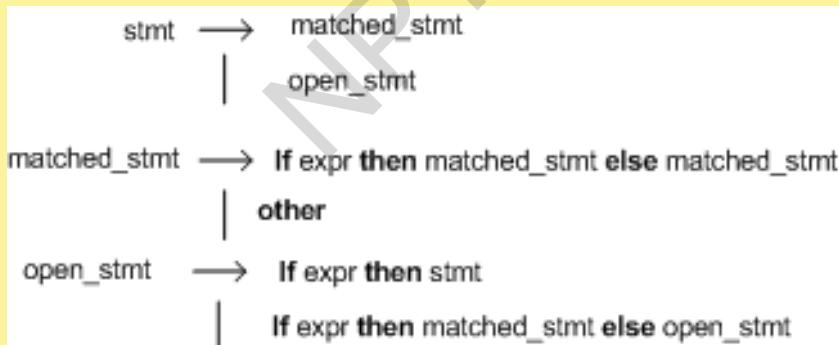


if E1 then S1 else if E2 then S2 else S3



Elimination of ambiguity (cont.)

- Idea:
 - A statement appearing between a **then** and an **else** must be matched



Elimination of left recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow A\alpha$
- Top down parsing methods can't handle left-recursive grammars
- A simple rule for direct left recursion elimination:
 - For a rule like:
 - $A \rightarrow A\alpha \mid \beta$
 - We may replace it with
 - $A \rightarrow \beta \ A'$
 - $A' \rightarrow \alpha \ A' \mid \epsilon$



Left recursion elimination (cont.)

- There are cases like following
 - $S \rightarrow Aa \mid b$
 - $A \rightarrow Ac \mid Sd \mid \epsilon$
- Left recursion elimination algorithm:
 - Arrange the nonterminals in some order A_1, A_2, \dots, A_n .
 - For (each i from 1 to n) {
 For (each j from 1 to $i-1$) {
 Replace each production of the form $A_i \rightarrow A_j \gamma$ by the production $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all current A_j productions
 }
 Eliminate left recursion among the A_i -productions
}



Left Recursion Elimination Example

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

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


Left factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.
- Consider following grammar:
 - Stmt \rightarrow if expr **then** stmt **else** stmt
 - | if expr **then** stmt
- On seeing input **if** it is not clear for the parser which production to use
- We can easily perform left factoring:
 - If we have $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$ then we replace it with
 - $A \rightarrow \alpha A'$
 - $A' \rightarrow \beta_1 \mid \beta_2$



Left factoring (cont.)

- Algorithm
 - For each non-terminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$, then replace all of A-productions $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid \gamma$ by
 - $A \rightarrow \alpha A' \mid \gamma$
 - $A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$
- Example:
 - $S \rightarrow i E t S \mid i E t S e S \mid a$
 - $E \rightarrow b$
- Modifies to
 - $S \rightarrow i E t S S' \mid a$
 - $S' \rightarrow e S \mid \epsilon$
 - $E \rightarrow b$



NPTEL

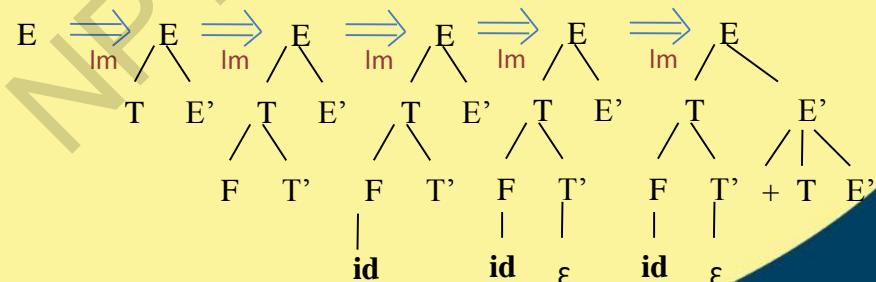
TOP DOWN PARSING



Introduction

- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right
- It can be also viewed as finding a leftmost derivation for an input string
- Example: $\text{id} + \text{id} * \text{id}$

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow (E) \mid \text{id}$



Recursive descent parsing

- Consists of a set of procedures, one for each nonterminal
- Execution begins with the procedure for start symbol
- A typical procedure for a non-terminal

```
void A() {  
    choose an A-production, A->X1X2..Xk  
    for (i = 1 to k) {  
        if (Xi is a nonterminal)  
            call procedure Xi();  
        else if (Xi equals the current input symbol a)  
            advance the input to the next symbol;  
        else /* an error has occurred */  
    }  
}
```



Recursive descent parsing (cont)

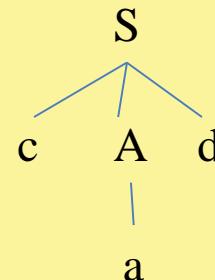
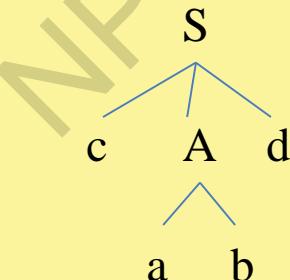
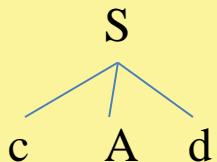
- General recursive descent may require backtracking
- The previous code needs to be modified to allow backtracking
- In general form it cannot choose an appropriate production easily.
- So we need to try all alternatives
- If one fails, the input pointer needs to be reset and another alternative has to be tried
- Recursive descent parsers cannot be used for left-recursive grammars



Example

$S \rightarrow cAd$
 $A \rightarrow ab \mid a$

Input: cad



Predictive parser

- It is a recursive-descent parser that needs no backtracking
- Suppose $A \rightarrow A_1 | A_2 | \dots | A_n$
- If the non-terminal to be expanded next is 'A', then the choice of rule is made on the basis of the current input symbol 'a'.



Procedure

- Make a **transition diagram** (like dfa/nfa) for every rule of the grammar.
- **Optimize** the dfa by reducing the number of states, yielding the final transition diagram
- To parse a string, **simulate** the string on the transition diagram
- If after consuming the input the transition diagram reaches an **accept state**, it is parsed.



Example

Consider the grammar:

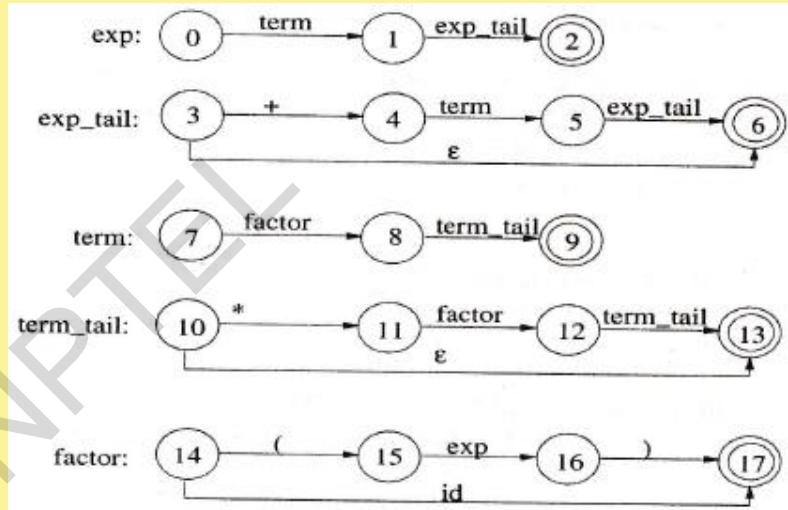
$\text{exp} \rightarrow \text{term exp_tail}$

$\text{exp_tail} \rightarrow + \text{term exp_tail} \mid \epsilon$

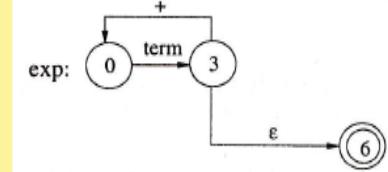
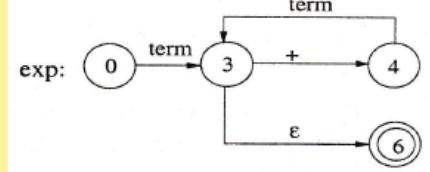
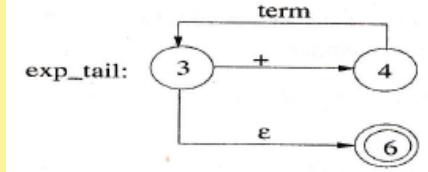
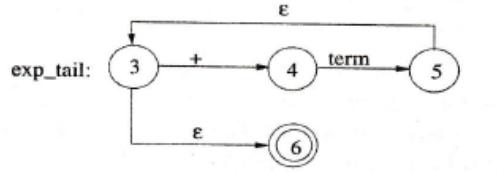
$\text{term} \rightarrow \text{factor term_tail}$

$\text{term_tail} \rightarrow * \text{factor term_tail} \mid \epsilon$

$\text{factor} \rightarrow (\text{exp}) \mid \text{id}$



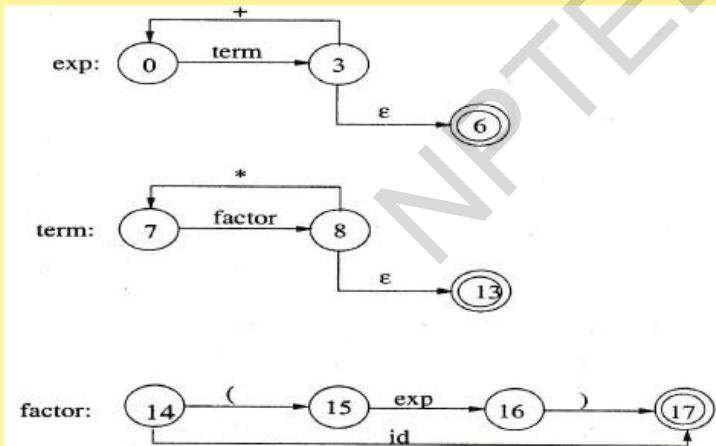
Example – Simplification



Eliminate self-recursion

Remove redundant ϵ edge

Substituting exp_tail into exp



Final set of diagrams



SIMULATION METHOD

- Start from the start state
- If a **terminal** comes **consume** it, move to next state
- If a **non – terminal** comes go to the state of the “dfa” of the non-term and return on reaching the final state
- Return to the original “dfa” and continue parsing
- If on completion(**reading input string completely**), you reach a final state, string is successfully parsed.



Disadvantage

- It is inherently a recursive parser, so it consumes a lot of memory as the stack grows.
- To remove this recursion, we use LL-parser, which uses a table for lookup.



First and Follow

- **First(α)** is set of terminals that begins strings derived from α
- If $\alpha \stackrel{*}{\Rightarrow} \varepsilon$ then ε is also in First(α)
- In predictive parsing when we have $A \rightarrow \alpha | \beta$, if First(α) and First(β) are disjoint sets then we can select appropriate A-production by looking at the next input
- **Follow(A)**, for any nonterminal A, is set of terminals a that can appear immediately after A in some sentential form
 - If we have $S \stackrel{*}{\Rightarrow} \alpha A a \beta$ for some α and β then a is in Follow(A)
- If A can be the rightmost symbol in some sentential form, then \$ is in Follow(A)



Computing First

- To compute First(X), apply following rules until no more terminals or ϵ can be added to any First set:
 1. If X is a terminal then $\text{First}(X) = \{X\}$.
 1. If X is a nonterminal and $X \rightarrow Y_1Y_2\dots Y_k$ is a production for some $k \geq 1$, then place a in $\text{First}(X)$ if for some i a is in $\text{First}(Y_i)$ and ϵ is in all of $\text{First}(Y_1), \dots, \text{First}(Y_{i-1})$ that is $Y_1\dots Y_{i-1} \Rightarrow \epsilon$. If ϵ is in $\text{First}(Y_j)$ for $j=1,\dots,k$ then add ϵ to $\text{First}(X)$.*
 1. If $X \rightarrow \epsilon$ is a production then add ϵ to $\text{First}(X)$



Computing Follow

- To compute $\text{Follow}(A)$ for all nonterminals A, apply following rules until nothing can be added to any follow set:
 1. Place $\$$ in $\text{Follow}(S)$ where S is the start symbol
 2. If there is a production $A \rightarrow \alpha B \beta$ then everything in $\text{First}(\beta)$ except ϵ is in $\text{Follow}(B)$.
 3. If there is a production $A \rightarrow \alpha B$ or a production $A \rightarrow \alpha B \beta$ where $\text{First}(\beta)$ contains ϵ , then everything in $\text{Follow}(A)$ is in $\text{Follow}(B)$



Example of First and Follow Sets

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

	First	Follow
F	{(,id}	{+, *,), \$}
T	{(,id}	{+,), \$}
E	{(,id}	{), \$}
E'	{+,ε}	{), \$}
T'	{*,ε}	{+,), \$}



LL(1) Grammars

- Predictive parsers are those recursive descent parsers needing no backtracking
- Grammars for which we can create predictive parsers are called LL(1)
 - The first L means scanning input from left to right
 - The second L means leftmost derivation
 - And 1 stands for using one input symbol for lookahead
 - More general one is LL(k), with k symbol lookahead
- A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
 - For no terminal a do α and β both derive strings beginning with a
 - At most one of α or β can derive empty string
 - If $\alpha \Rightarrow^* \varepsilon$ then β does not derive any string beginning with a terminal in $\text{Follow}(A)$



Construction of predictive parsing table

- For each production $A \rightarrow \alpha$ in grammar do the following:
 1. For each terminal a in $\text{First}(\alpha)$ add $A \rightarrow \alpha$ in $M[A,a]$
 2. If ϵ is in $\text{First}(\alpha)$, then for each terminal b in $\text{Follow}(A)$ add $A \rightarrow \epsilon$ to $M[A,b]$. If ϵ is in $\text{First}(\alpha)$ and $\$$ is in $\text{Follow}(A)$, add $A \rightarrow \epsilon$ to $M[A,\$]$ as well
- If after performing the above, there is no production in $M[A,a]$ then set $M[A,a]$ to error



Example

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

	First	Follow
F	$\{(, id\}$	$\{+, *,), \$\}$
T	$\{(, id\}$	$\{+,), \$\}$
E	$\{(, id\}$	$\{\), \$\}$
E'	$\{+, \epsilon\}$	$\{\), \$\}$
T'	$\{*, \epsilon\}$	$\{+,), \$\}$

Non-terminal	Input Symbol					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'			$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		



Another example

$S \rightarrow iEtSS' \mid a$
 $S' \rightarrow eS \mid \epsilon$
 $E \rightarrow b$

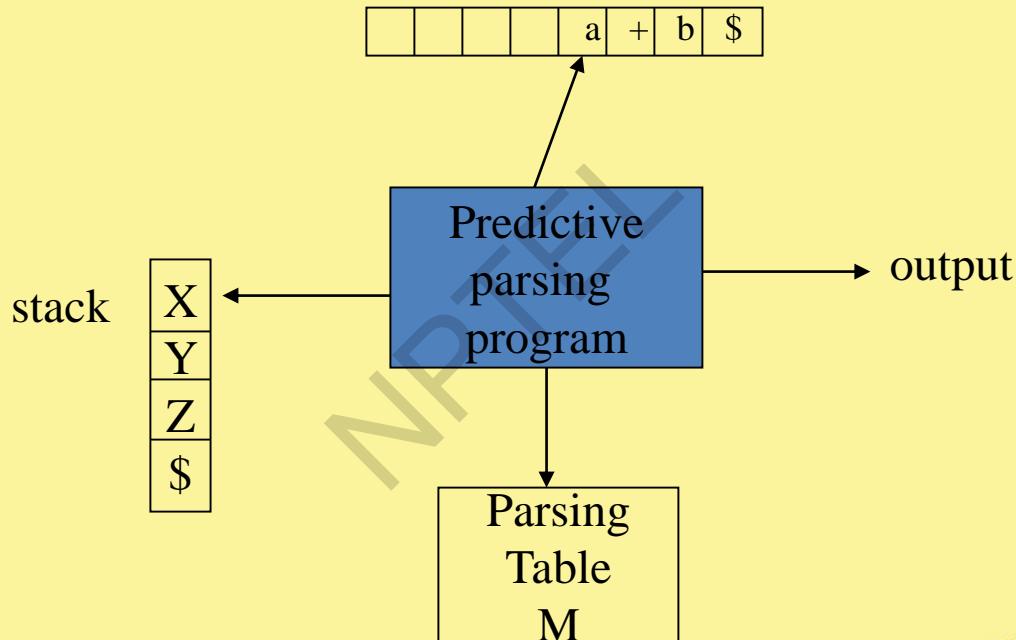
$\text{First}(S) = \{i, a\}$
 $\text{First}(S') = \{e, \epsilon\}$
 $\text{First}(E) = \{b\}$

$\text{Follow}(S) = \{\$, e\}$
 $\text{Follow}(S') = \{\$, e\}$
 $\text{Follow}(E) = \{t\}$

Non-terminal	Input Symbol					
	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				



Non-recursive predicting parsing



Predictive parsing algorithm

While (stack is not empty) do

 Let X be the top symbol in the stack;

 Let a be the next input symbol;

 if (X is a) pop the stack and advance input pointer;

 else if (X is a terminal) error();

 else if ($M[X,a]$ is an error entry) error();

 else if ($M[X,a] = X \rightarrow Y_1 Y_2 \dots Y_k$) {

 output the production $X \rightarrow Y_1 Y_2 \dots Y_k$;

 pop the stack;

 push Y_k, \dots, Y_2, Y_1 on to the stack with Y_1 on top;

}



Example

$\text{id} + \text{id} * \text{id} \$$

Stack	Input	Action
E	$\text{id} + \text{id} * \text{id} \$$	Parse $E \rightarrow TE'$
$E'T$	$\text{id} + \text{id} * \text{id} \$$	Parse $E' \rightarrow FT'$
$E'T'F$	$\text{id} + \text{id} * \text{id} \$$	Parse $F \rightarrow id$
$E'T'id$	$\text{id} + \text{id} * \text{id} \$$	Advance input
$E'T'$	$+ \text{id} * \text{id} \$$	Parse $T' \rightarrow \epsilon$
E'	$+ \text{id} * \text{id} \$$	Parse $E' \rightarrow +TE'$
$E'T+$	$+ \text{id} * \text{id} \$$	Advance input
$E'T$	$\text{id} * \text{id} \$$	Parse $T \rightarrow FT'$

Stack	Input	Action
$E'T'F$	$\text{id} * \text{id} \$$	Parse $F \rightarrow id$
$E'T'id$	$\text{id} * \text{id} \$$	Advance input
$E'T'$	$* \text{id} \$$	Parse $T' \rightarrow *FT'$
$E'T'F^*$	$* \text{id} \$$	Advance input
$E'T'F$	$\text{id} \$$	Parse $F \rightarrow id$
$E'T'id$	$\text{id} \$$	Advance input
$E'T'$	$\$$	Parse $T' \rightarrow \epsilon$
E'	$\$$	Parse $E' \rightarrow \epsilon$
	$\$$	Done

Non-terminal	Input Symbol					
	id	+	*	()	\$
E	$E \rightarrow TE'$				$E \rightarrow TE'$	
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$				$T \rightarrow FT'$	
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$				$F \rightarrow (E)$	

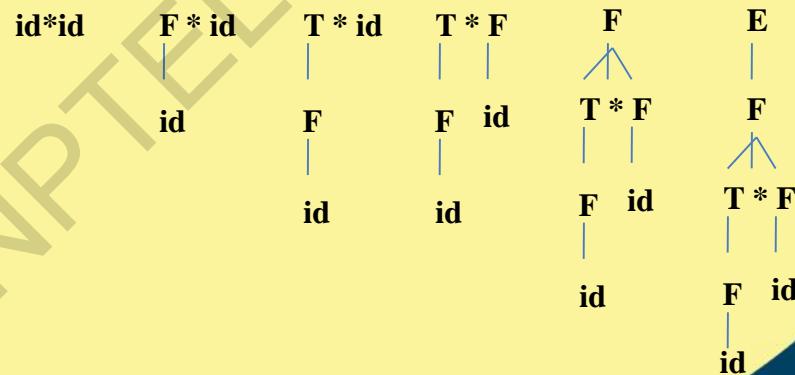


Bottom-up Parsing



Introduction

- Constructs parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Example: id*id

$$\begin{aligned}E &\rightarrow E + T \mid T \\T &\rightarrow T * F \mid F \\F &\rightarrow (E) \mid id\end{aligned}$$


Shift-reduce parser

- The general idea is to shift some symbols of input to the stack until a reduction can be applied
- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of the production
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply
- A reduction is a reverse of a step in a derivation
- The goal of a bottom-up parser is to construct a derivation in reverse:
 - $E \Rightarrow T \Rightarrow T^* F \Rightarrow T^* id \Rightarrow F^* id \Rightarrow id^* id$



Handle pruning

- A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation

Right sentential form	Handle	Reducing production
id^*id	id	$F \rightarrow id$
F^*id	F	$T \rightarrow F$
T^*id	id	$F \rightarrow id$
T^*F	T^*F	$E \rightarrow T^*F$



Shift reduce parsing

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:

Stack	Input
\$	w\$

- Acceptance configuration

Stack	Input
\$S	\$



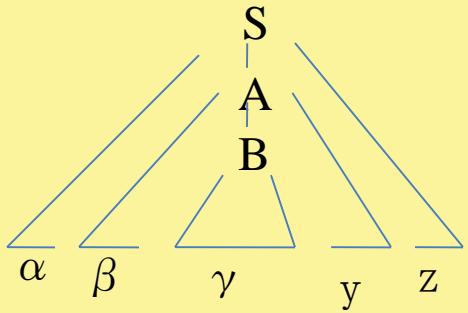
Shift reduce parsing (cont.)

- Basic operations:
 - Shift
 - Reduce
 - Accept
 - Error
- Example: $\text{id}^* \text{id}$

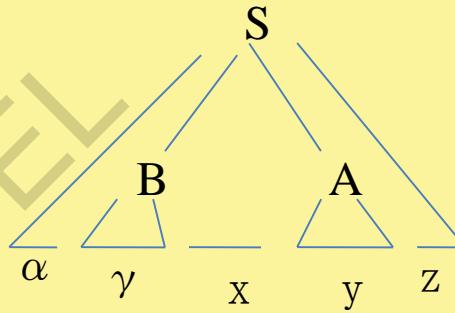
Stack	Input	Action
\$	$\text{id}^* \text{id} \$$	shift
$\$ \text{id}$	$* \text{id} \$$	reduce by $F \rightarrow \text{id}$
$\$ F$	$* \text{id} \$$	reduce by $T \rightarrow F$
$\$ T$	$* \text{id} \$$	shift
$\$ T^*$	$\text{id} \$$	shift
$\$ T^* \text{id}$	\$	reduce by $F \rightarrow \text{id}$
$\$ T^* F$	\$	reduce by $T \rightarrow T^* F$
$\$ T$	\$	reduce by $E \rightarrow T$
$\$ E$	\$	accept



Handle will appear on top of the stack



Stack	Input
$\$ \alpha \beta \gamma$	yz\$
$\$ \alpha \beta B$	yz\$
$\$ \alpha \beta By$	z\$



Stack	Input
$\$ \alpha \gamma$	xyz\$
$\$ \alpha Bxy$	z\$



Conflicts during shift reduce parsing

- Two kind of conflicts
 - Shift/reduce conflict
 - Reduce/reduce conflict
- Example:

stmt \rightarrow If expr then stmt
| If expr then stmt else stmt
| other

Stack Input
... if expr then stmt else ...\$



Reduce/reduce conflict

stmt -> id(parameter_list)

stmt -> expr:=expr

parameter_list->parameter_list, parameter

parameter_list->parameter

parameter->id

expr->id(expr_list)

expr->id

expr_list->expr_list, expr

expr_list->expr

Stack
... id(id

Input
,id) ...\$



Bottom-Up Parsing

- Operator Precedence Parsing
- LR Parsing

NPTEL



Operator Precedence Parsing



Operator Grammar

- No ϵ -transition
- No two adjacent non-terminals

Eg.

$$E \rightarrow E \text{ op } E \mid \text{id}$$

$$\text{op} \rightarrow + \mid *$$

The above grammar is not an operator grammar
but:

$$E \rightarrow E + E \mid E^* E \mid \text{id}$$



Operator Precedence

- If a has higher precedence over b; $a \cdot> b$
- If a has lower precedence over b; $a <\cdot b$
- If a and b have equal precedence; $a \doteq b$

Note:

- id has higher precedence than any other symbol
- \$ has lowest precedence.
- if two operators have equal precedence, then we check the **Associativity** of that particular operator.



Precedence Table

	id	+	*	\$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
\$	<·	<·	<·	·>

Example: $w = \$id + id * id\$$
 $\$ <\cdot id \cdot> + <\cdot id \cdot> * <\cdot id \cdot> \$$



Basic Principle

- Scan input string left to right, try to detect $\cdot >$ and put a pointer on its location.
- Now scan backwards till reaching $< \cdot$.
- String between $< \cdot$ and $\cdot >$ is our handle.
- Replace handle by the head of the respective production.
- REPEAT until reaching start symbol.



Algorithm

```
Initialize stack to $  
while true do  
    let  $U$  be the topmost terminal in the stack  
    let  $V$  be the next input symbol  
    if  $U = \$$  and  $V = \$$  then return  
    if  $U \prec V$  or  $U \doteq V$  then  
        shift  $V$  onto stack  
        advance input pointer  
    else if  $U \cdot > V$  then  
        do  
            pop the topmost symbol, call it  $V$ , from the stack  
            until the top of the stack is  $\prec V$   
    else  
        error  
end
```



Example

STACK	INPUT	ACTION/REMARK
\$	id + id * id\$	\$ <· id
\$ id	+ id * id\$	id ·> +
\$	+ id * id\$	\$ <· +
\$ +	id * id\$	+ <· id
\$ + id	* id\$	id ·> *
\$ +	* id\$	+ <· *
\$ + *	id\$	* <· id
\$ + * id	\$	id ·> \$
\$ + *	\$	* ·> \$
\$ +	\$	+ ·> \$
\$	\$	accept

	id	+	*	\$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
\$	<·	<·	<·	·>



Establishing Precedence Relationships

- Construct two lists: Firstop+, Lastop+
- Firstop+: List of all terminals which can appear first on any right hand side of a production
- Lastop+: List of terminals that can appear last on any right hand side of a production



Firstop and Lastop

- For $X \rightarrow a\dots | Bc$, put a, B, c in $\text{Firstop}(X)$
- For $Y \rightarrow \dots u | \dots vW$, put u, v, W in $\text{Lastop}(Y)$
- Compute Firstop^+ and Lastop^+ using Closure algorithm
 - Take each nonterminal in turn, in any order and look for it in all the Firstop lists. Add its own first symbol list to any other in which it occurs
 - Similarly process Lastop list
 - Drop all nonterminals from the lists



Constructing Precedence Matrix

- Whenever terminal a immediately precedes nonterminal B in any production, put $a < \cdot \alpha$ where α is any terminal in the First $_{\text{top+}}$ list of B
- Whenever terminal b immediately follows nonterminal C in any production, put $\beta \cdot > b$ where β is any terminal in the Last $_{\text{top+}}$ list of C
- Whenever a sequence aBc or ac occurs in any production, put $a \doteq c$
- Add relations $\$ < \cdot a$ and $a \cdot > \$$ for all terminals in the First $_{\text{top+}}$ and Last $_{\text{top+}}$ lists, respectively of S



Example

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$\text{Firststop}(E) = \{E, +, T\}$

$\text{Firststop}(T) = \{T, *, F\}$

$\text{Firststop}(F) = \{(), id\}$

$\text{Laststop}(E) = \{+, T\}$

$\text{Laststop}(T) = \{*, F\}$

$\text{Laststop}(F) = \{(), id\}$

$\text{Firststop+}(E) = \{E, +, T, *, F, (), id\}$

$\text{Firststop+}(T) = \{T, *, F, (), id\}$

$\text{Firststop+}(F) = \{(), id\}$

$\text{Laststop+}(E) = \{+, T, *, F, (), id\}$

$\text{Laststop+}(T) = \{*, F, (), id\}$

$\text{Laststop+}(F) = \{(), id\}$

$\text{Firststop+}(E) = \{+, *, (), id\}$

$\text{Firststop+}(T) = \{*, (), id\}$

$\text{Firststop+}(F) = \{(), id\}$

$\text{Laststop+}(E) = \{+, *, (), id\}$

$\text{Laststop+}(T) = \{*, (), id\}$

$\text{Laststop+}(F) = \{(), id\}$



Example (Contd.)

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$\text{First}_{\text{top}}(E) = \{+, *, (, id\}$

$\text{First}_{\text{top}}(T) = \{*, (, id\}$

$\text{First}_{\text{top}}(F) = \{(, id\}$

$\text{Last}_{\text{top}}(E) = \{+, *,), id\}$

$\text{Last}_{\text{top}}(T) = \{*,), id\}$

$\text{Last}_{\text{top}}(F) = \{), id\}$

	\$	()	id	+	*
\$		<·		<·	<·	<·
(<·	≡	<·	<·	<·
)	·>		·>		·>	·>
id	·>		·>		·>	·>
+	·>	<·	·>	<·	·>	<·
*	·>	<·	·>	<·	·>	·>



Example (Contd.)

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$\text{Firststop}(E) = \{+, *, (, id\}$

$\text{Firststop}(T) = \{*, (, id\}$

$\text{Firststop}(F) = \{(, id\}$

$\text{Laststop}(E) = \{+, *,), id\}$

$\text{Laststop}(T) = \{*,), id\}$

$\text{Laststop}(F) = \{), id\}$

	\$	()	id	+	*
\$		<·		<·	<·	<·
(<·	≡	<·	<·	<·
)	·>		·>		·>	·>
id	·>		·>		·>	·>
+	·>	<·	·>	<·	·>	<·
*	·>	<·	·>	<·	·>	·>



LR Parsing

- The most prevalent type of bottom-up parsers
- LR(k), mostly interested on parsers with $k \leq 1$
- Why LR parsers?
 - Table driven
 - Can be constructed to recognize all programming language constructs
 - Most general non-backtracking shift-reduce parsing method
 - Can detect a syntactic error as soon as it is possible to do so
 - Class of grammars for which we can construct LR parsers are superset of those which we can construct LL parsers



LR Parsing Methods

- SLR – Simple LR. Easy to implement, less powerful
- Canonical LR – most general and powerful. Tedious and costly to implement, contains much more number of states compared to SLR
- LALR – Look Ahead LR. Mix of SLR and Canonical LR. Can be implemented efficiently, contains same number of states as simple LR for a grammar



States of an LR parser

- States represent set of items
- An LR(0) item of G is a production of G with the dot at some position of the body:
 - For $A \rightarrow XYZ$ we have following items
 - $A \rightarrow .XYZ$
 - $A \rightarrow X.YZ$
 - $A \rightarrow XY.Z$
 - $A \rightarrow XYZ.$
 - In a state having $A \rightarrow .XYZ$ we hope to see a string derivable from XYZ next on the input.
 - What about $A \rightarrow X.YZ$?



Constructing canonical LR(0) item sets

- Augmented grammar:
 - G with addition of a production: $S' \rightarrow S$
- Closure of item sets:
 - If I is a set of items, $\text{closure}(I)$ is a set of items constructed from I by the following rules:
 - Add every item in I to $\text{closure}(I)$
 - If $A \rightarrow \alpha.B\beta$ is in $\text{closure}(I)$ and $B \rightarrow \gamma$ is a production then add the item $B \rightarrow .\gamma$ to $\text{closure}(I)$.
- Example:

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

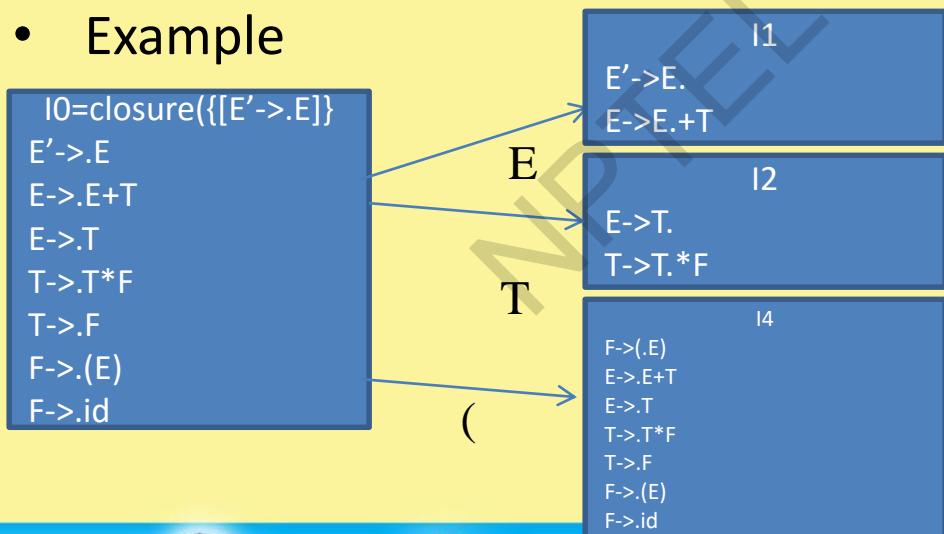
$I_0 = \text{closure}(\{[E' \rightarrow .E]\})$

$E' \rightarrow .E$
 $E \rightarrow .E + T$
 $E \rightarrow .T$
 $T \rightarrow .T * F$
 $T \rightarrow .F$
 $F \rightarrow .(E)$
 $F \rightarrow .id$



Constructing canonical LR(0) item sets (cont.)

- Goto (I, X) where I is an item set and X is a grammar symbol is closure of set of all items $[A \rightarrow \alpha X. \beta]$ where $[A \rightarrow \alpha X \beta]$ is in I
- Example



Closure algorithm

```
SetOfItems CLOSURE(I) {  
    J=I;  
    repeat  
        for (each item A-> α.Bβ in J)  
            for (each production B->γ of G)  
                if (B->.γ is not in J)  
                    add B->.γ to J;  
    until no more items are added to J on one round;  
    return J;  
}
```



GOTO algorithm

```
SetOfItems GOTO(I,X) {  
    J=empty;  
    if (A-> α.X β is in I)  
        add CLOSURE(A-> αX. β ) to J;  
    return J;  
}
```



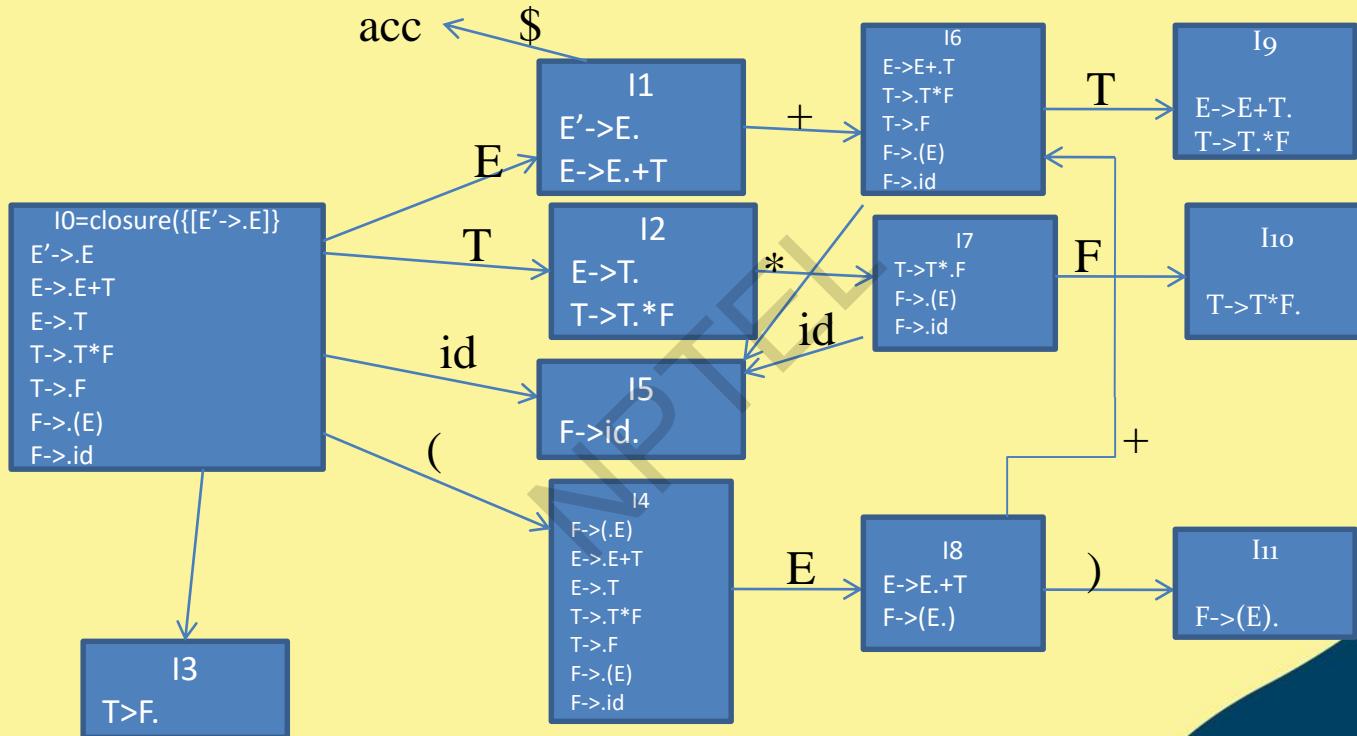
LR(0) items

```
Void items(G') {  
    C= CLOSURE({[S'->.S]});  
    repeat  
        for (each set of items I in C)  
            for (each grammar symbol X)  
                if (GOTO(I,X) is not empty and not in C)  
                    add GOTO(I,X) to C;  
    until no new set of items are added to C on a round;  
}
```



Example

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



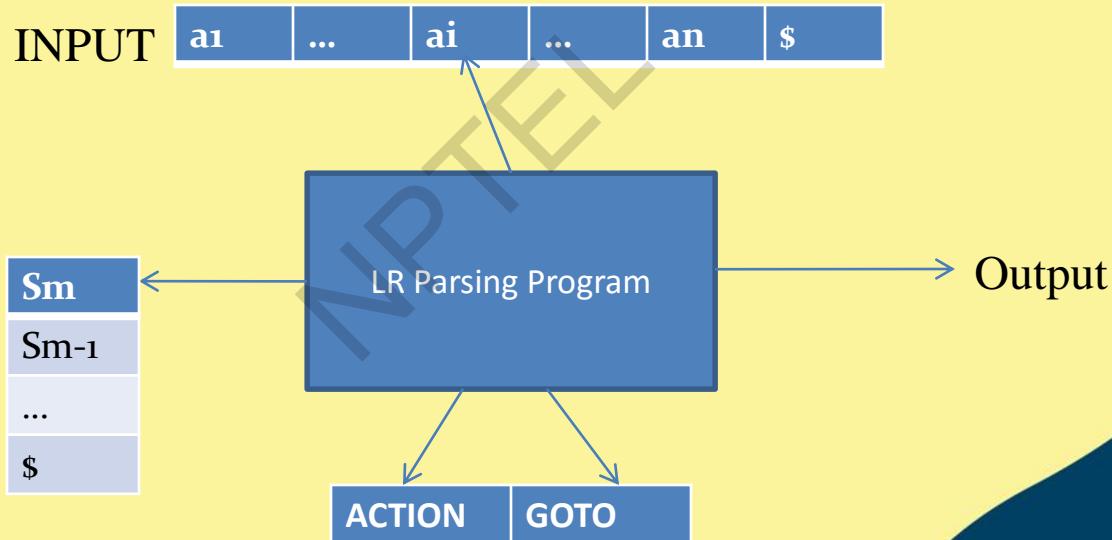
Use of LR(0) automaton

- Example: $\text{id}^* \text{id}$

Line	Stack	Symbols	Input	Action
(1)	o	\$	$\text{id}^* \text{id} \$$	Shift to 5
(2)	o5	\$id	$*\text{id} \$$	Reduce by $F \rightarrow \text{id}$
(3)	o3	\$F	$*\text{id} \$$	Reduce by $T \rightarrow F$
(4)	o2	\$T	$*\text{id} \$$	Shift to 7
(5)	o27	\$T*	id\$	Shift to 5
(6)	o275	\$T*id	\$	Reduce by $F \rightarrow \text{id}$
(7)	o2710	\$T*T	\$	Reduce by $T \rightarrow T^*F$
(8)	o2	\$T	\$	Reduce by $E \rightarrow T$
(9)	o1	\$E	\$	accept



LR-Parsing model



LR parsing algorithm

```
let a be the first symbol of w$;  
while(1) { /*repeat forever */  
    let s be the state on top of the stack;  
    if (ACTION[s,a] = shift t) {  
        push t onto the stack;  
        let a be the next input symbol;  
    } else if (ACTION[s,a] = reduce A->β) {  
        pop |β| symbols of the stack;  
        let state t now be on top of the stack;  
        push GOTO[t,A] onto the stack;  
        output the production A->β;  
    } else if (ACTION[s,a]=accept) break; /* parsing is done */  
    else call error-recovery routine;  
}
```



Example

STATE	ACTION					GOTO			
	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			

Line	Stack	Symbols	Input	Action
(1)	0		id*id+id\$	Shift to 5
(2)	05	id	*id+id\$	Reduce by F->id
(3)	03	F	*id+id\$	Reduce by T->F
(4)	02	T	*id+id\$	Shift to 7
(5)	027	T*	id+id\$	Shift to 5
(6)	0275	T*id	+id\$	Reduce by F->id
(7)	02710	T*T	+id\$	Reduce by T->T*T
(8)	02	T	+id\$	Reduce by E->T
(9)	01	E	+id\$	Shift
(10)	016	E+	id\$	Shift
(11)	0165	E+id	\$	Reduce by F->id
(12)	0163	E+F	\$	Reduce by T->F
(13)	0169	E+T	\$	Reduce by E->E+T
(14)	01	E	\$	accept

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

id*id+id\$



Constructing SLR parsing table

- Method
 - Construct $C=\{I_0, I_1, \dots, I_n\}$, the collection of LR(0) items for G'
 - State i is constructed from state l_i :
 - If $[A \rightarrow \alpha.a\beta]$ is in l_i and $\text{Goto}(l_i, a) = l_j$, then set $\text{ACTION}[i, a]$ to “shift j ”
 - If $[A \rightarrow \alpha.]$ is in l_i , then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$ ” for all a in $\text{follow}(A)$
 - If $[S' \rightarrow S.]$ is in l_i , then set $\text{ACTION}[i, \$]$ to “Accept”
 - If any conflicts appears then we say that the grammar is not SLR(1).
 - If $\text{GOTO}(l_i, A) = l_j$ then $\text{GOTO}[i, A] = j$
 - All entries not defined by above rules are made “error”
 - The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow .S]$



Example grammar which is not SLR

$$S \rightarrow L=R \mid R$$
$$L \rightarrow *R \mid id$$
$$R \rightarrow L$$

I0
 $S' \rightarrow S$
 $S \rightarrow .L=R$
 $S \rightarrow .R$
 $L \rightarrow .*R \mid$
 $L \rightarrow .id$
 $R \rightarrow .L$

I1
 $S' \rightarrow S.$

I2
 $S \rightarrow L.=R$
 $R \rightarrow L.$

I3
 $S \rightarrow R.$

I4
 $L \rightarrow *.R$
 $R \rightarrow .L$
 $L \rightarrow .*R$
 $L \rightarrow .id$

I5
 $L \rightarrow id.$

I6
 $S \rightarrow L=.R$
 $R \rightarrow .L$
 $L \rightarrow .*R$
 $L \rightarrow .id$

I7
 $L \rightarrow *R.$

I8
 $R \rightarrow L.$

I9
 $S \rightarrow L=R.$

Action = Shift 6
Reduce $R \rightarrow L$



More powerful LR parsers

- Canonical-LR or just LR method
 - Use lookahead symbols for items: LR(1) items
 - Results in a large collection of items
- LALR: lookaheads are introduced in LR(0) items



LR(1) Grammar

- A grammar is said to be LR(1) if in a single left-to-right scan, we can construct a reverse rightmost derivation, while using atmost a single token lookahead to resolve ambiguities.
- LR(k) parsers use k token lookahead



LR(k) item

- A pair $[\alpha; \beta]$, where
 - α is a production from G with a $.$ at some position in the right hand side
 - β is a lookahead string contains k symbols (terminals or $\$$)
- Several LR(1) items may have same core. $[A \rightarrow X.YZ;a]$ and $[A \rightarrow X.YZ;b]$ are represented together as $[A \rightarrow X.YZ;\{a,b\}]$



Usage of LR(1) Lookahead

- Carry them along to allow choosing correct reduction when there is any choice
- Lookaheads are bookkeeping, unless item has a . at right end
 - In $[A \rightarrow X.YZ;a]$, a has no direct use
 - In $[A \rightarrow XYZ.;a]$, a is useful
 - If there are two items $[A \rightarrow XYZ.;a]$ and $[B \rightarrow XYZ.;b]$, we can decide between reducing to A or B by looking at limited right context



Closure algorithm

```
SetOfItems CLOSURE(I) {  
    J=I;  
    repeat  
        for (each item [A-> α.Bβ;a] in J)  
            for (each production B->γ of G and each terminal b in First(βa)  
                if ([B->.γ;b] is not in J)  
                    add [B->.γ;b] to J;  
    until no more items are added to J on one round;  
    return J;  
}
```



GOTO algorithm

```
SetOfItems GOTO(I,X) {
```

```
    J=empty;
```

```
    if ([A-> α.Xβ;a] is in I)
```

```
        add CLOSURE([A-> αX.β;a] ) to J;
```

```
    return J;
```

```
}
```



Constructing LR(1) Parsing Table

- Method
 - Construct $C=\{I_0, I_1, \dots, I_n\}$, the collection of LR(1) items for G'
 - State i is constructed from state I_i :
 - If $[A \rightarrow \alpha . a \beta; b]$ is in I_i and $\text{Goto}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to “shift j ”
 - If $[A \rightarrow \alpha .; a]$ is in I_i , then set $\text{ACTION}[i, a]$ to “reduce $A \rightarrow \alpha$ ”
 - If $[S' \rightarrow S .; \$]$ is in I_i , then set $\text{ACTION}[i, \$]$ to “Accept”
 - If any conflicts appears then we say that the grammar is not SLR(1).
 - If $\text{GOTO}(I_i, A) = I_j$ then $\text{GOTO}[i, A] = j$
 - All entries not defined by above rules are made “error”
 - The initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow .S; \$]$



Example LR(1) Parser

goal \rightarrow expr

expr \rightarrow term + expr

expr \rightarrow term

term \rightarrow factor * term

term \rightarrow factor

factor \rightarrow id

I_0 : $[goal \rightarrow \cdot expr, \$], [expr \rightarrow \cdot term + expr, \$], [expr \rightarrow \cdot term, \$], [term \rightarrow \cdot factor * term, \{+, \$\}], [term \rightarrow \cdot factor, \{+, \$\}], [factor \rightarrow id, \{+, *, \$\}]$

I_1 : $[goal \rightarrow expr \cdot, \$]$

I_2 : $[expr \rightarrow term \cdot, \$], [expr \rightarrow term \cdot + expr, \$]$

I_3 : $[term \rightarrow factor \cdot, \{+, \$\}], [term \rightarrow factor \cdot * term, \{+, \$\}]$

I_4 : $[factor \rightarrow id \cdot, \{+, *, \$\}]$

I_5 : $[expr \rightarrow term + \cdot expr, \$], [expr \rightarrow \cdot term + expr, \$], [expr \rightarrow \cdot term, \$], [term \rightarrow \cdot factor * term, \{+, \$\}], [term \rightarrow \cdot factor, \{+, \$\}], [factor \rightarrow \cdot id, \{+, *, \$\}]$

I_6 : $[term \rightarrow factor * \cdot term, \{+, \$\}], [term \rightarrow \cdot factor * term, \{+, \$\}], [term \rightarrow \cdot factor, \{+, \$\}], [factor \rightarrow \cdot id, \{+, *, \$\}]$

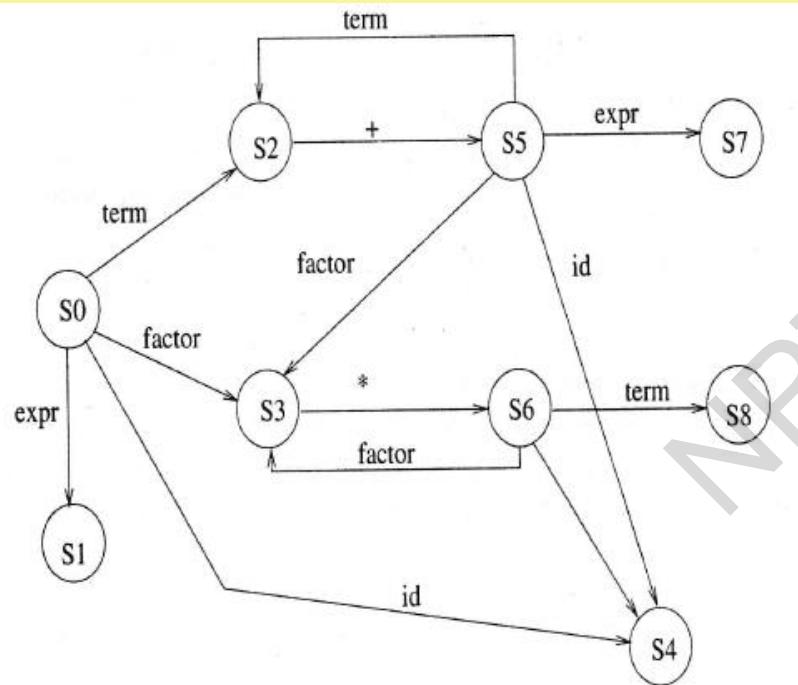
I_7 : $[expr \rightarrow term + \cdot expr, \$]$

I_8 : $[term \rightarrow factor * term \cdot, \{+, \$\}]$

LR(1) items



Example LR(1) Parser (Contd.)



							Expr	Term	factor
	id	+	*	\$					
0	S4						1	2	3
1						Acc			
2			S5			R3			
3		S5	R6	R5					
4		R6	R6	R6					
5	S4					7	2	3	
6	S4						8	3	
7						R2			
8			R4			R4			



LALR(1) Parsing

- Reduces number of states in an LR(1) parser
- Merges states differing only in lookahead sets
- SLR and LALR tables have same number of states
- For a C-like language, several hundred states in SLR and LALR parsers, several thousands for LR(1)



Example

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC \mid d$

LR(1) items

$I_0 : [S' \rightarrow \cdot S, \$], [S \rightarrow \cdot CC, \$], [C \rightarrow \cdot cC, \{c, d\}], [C \rightarrow \cdot d, \{c, d\}]$

$I_1 : [S' \rightarrow S \cdot, \$]$

$I_2 : [S \rightarrow C \cdot C, \$], [C \rightarrow \cdot cC, \$], [C \rightarrow \cdot d, \$]$

$I_3 : [C \rightarrow c \cdot C, \{c, d\}], [C \rightarrow \cdot cC, \{c, d\}], [C \rightarrow \cdot d, \{c, d\}]$

$I_4 : [C \rightarrow d \cdot, \{c, d\}]$

$I_5 : [C \rightarrow CC \cdot, \$]$

$I_6 : [C \rightarrow c \cdot C, \$], [C \rightarrow \cdot cC, \$], [C \rightarrow \cdot d, \$]$

$I_7 : [C \rightarrow d \cdot, \$]$

$I_8 : [C \rightarrow cC \cdot, \{c, d\}]$

$I_9 : [C \rightarrow cC \cdot, \$]$

States I_4 and I_7 , I_3 and I_6 , I_8 and I_9 can be merged

$I_{47} : [C \rightarrow d \cdot ; \{c, d, \$\}]$

$I_{36} : [C \rightarrow c.C ; \{c, d, \$\}], [C \rightarrow .c ; \{c, d, \$\}], [C \rightarrow .d ; \{c, d, \$\}]$

$I_{89} : [C \rightarrow cC \cdot ; \{c, d, \$\}]$



LALR Construction – Step-by-Step Approach

- Sets of states constructed as in LR(1) method
- At each point where a new set is spawned, it may be merged with an existing set
- When a new state S is created, all other states are checked to see if one with the same core exists
- If not, S is kept; otherwise it is merged with the existing set T with the same core to form state ST



Using Ambiguous Grammars

$E \rightarrow E+E$
 $E \rightarrow E^*E$
 $E \rightarrow (E)$
 $E \rightarrow id$

$\text{Follow}(E) = \{+, *, (), \$\}$

				I6: $E \rightarrow (E.)$
I0: $E' \rightarrow .E$	I2: $E \rightarrow (.E)$	I4: $E \rightarrow E+E.E$		$E \rightarrow E.+E$
$E \rightarrow .E+E$	$E \rightarrow .E+E$	$E \rightarrow .E+E$	I7: $E \rightarrow E+E.E$	$E \rightarrow E.*E$
$E \rightarrow .E^*E$	$E \rightarrow .E^*E$	$E \rightarrow .E^*E$	$E \rightarrow E.+E$	
$E \rightarrow .(E)$	$E \rightarrow .(E)$	$E \rightarrow .(E)$	$E \rightarrow E.+E$	
$E \rightarrow .id$	$E \rightarrow .id$	$E \rightarrow .id$	$E \rightarrow E.*E$	
I1: $E' \rightarrow E.$	I3: $E \rightarrow id.$	I5: $E \rightarrow E^*.E$	I8: $E \rightarrow E^*E.$	
$E \rightarrow E.+E$		$E \rightarrow (.E)$	$E \rightarrow E.+E$	
$E \rightarrow E.*E$		$E \rightarrow .E+E$	$E \rightarrow E.*E$	
		$E \rightarrow .E^*E$		
		$E \rightarrow .(E)$	I9: $E \rightarrow (E).$	
		$E \rightarrow .id$		

STATE	ACTION						GO TO
	id	+	*	()	\$	
0	S_3				S_2		1
1		S_4	S_5			Acc	
2	S_3		S_2				6
3		R_4	R_4		R_4	R_4	
4	S_3			S_2			7
5	S_3			S_2			8
6		S_4	S_5				
7		R_1/S_4	S_5/R_1		R_1	R_1	
8		R_2/S_4	R_2/S_5		R_2	R_2	
9		R_3	R_3		R_3	R_3	



Error Recovery in LR Parsing

- Undefined entries in LR parsing table means error
- Proper error messages can be flashed to the user
- Error handling routines can be made to modify the parser stack by
 - popping out some entries
 - pushing some desirable entries into the stack
- Brings parser at a descent stage from which it can proceed further
- Enables detection of multiple errors and flashing them to the user for correction



Error Recovery – Example

$E' \rightarrow E$

$E \rightarrow E + E \mid E * E \mid id$

$Follow(E) = \{+, *, \$\}$

I0: $\{[E' \rightarrow .E], [E \rightarrow .E+E], [E \rightarrow .E*E], [E \rightarrow .id]\}$

I1: $\{[E' \rightarrow E.], [E \rightarrow E.+E], [E \rightarrow E.*E]\}$

I2: $\{[E \rightarrow id.] \}$

I3: $\{[E \rightarrow E+E.], [E \rightarrow .E+E], [E \rightarrow .E*E], [E \rightarrow .id]\}$

I4: $\{[E \rightarrow E*.E], [E \rightarrow .E+E], [E \rightarrow .E*E], [E \rightarrow .id]\}$

I5: $\{[E \rightarrow E+E.], [E \rightarrow E.+E], [E \rightarrow E.*E]\}$

I6: $\{E \rightarrow E+E.], [E \rightarrow E.+E], [E \rightarrow E.*E]\}$

I7: $\{[E \rightarrow E*E.], [E \rightarrow E.+E], [E \rightarrow E.*E]\}$

State	ACTION					GOTO
	id	+	*	\$	E	
0	S2	e1	e1	e1		1
1	e2	S3	S4	Acc		
2	e2	R3	R3	R3		
3	S2	e1	e1	e1		5
4	S2	e1	e1	e1		6
5	e2	R1	S4	R1		
6	e2	R2	R2	R2		

e1: Seen operator or end of string
while expecting id

e2: Seen id while expecting
operator



Example (Contd.)

String: "id + *\$"

Stack	Input	Error message and action
0	id+*\$	Shift
0id2	+*\$	Reduce by E->id
0E1	+*\$	Shift
0E1+3	*\$	"id expected", pushed id and 2 to stack
0E1+3id2	*\$	Reduce by E->id
0E1+3E5	*\$	Shift
0E1+3E5*4	\$	"id expected", pushed id and 2 to stack
0E1+3E5*4id2	\$	Reduce by E->id
0E1+3E5*4E6	\$	Reduce by E->E*E
0E1+3E5	\$	Reduce by E->E+E
0E1	\$	Accept

State	ACTION					GOTO
	id	+	*	\$	E	
0	S2	e1	e1	e1	1	
1	e2	S3	S4	Acc		
2	e2	R3	R3	R3		
3	S2	e1	e1	e1	5	
4	S2	e1	e1	e1	6	
5	e2	R1	S4	R1		
6	e2	R2	R2	R2		



LALR Parser Generator - yacc

- yacc – Yet Another Compiler Compiler
- Automatically generates LALR parser for a grammar from its specification
- Input is divided into three sections
 - ...definitions... Consists of token declarations C code within %{ and %}
 - %%
 - ...rules... Contains grammar rules
 - %%
 - ...subroutines... Contains user subroutines



Example – Calculator to Add and Subtract Numbers

- Definition section

```
%token INTEGER
```

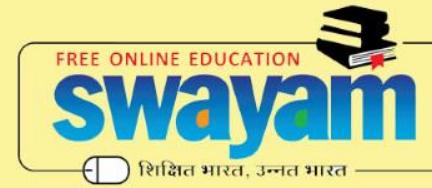
declares an INTEGER token

- Running yacc generates y.tab.c and y.tab.h files
- y.tab.h:

```
#ifndef YYSTYPE  
#define YYSTYPE int  
#endif  
#define INTEGER 258  
extern YYSTYPE yylval;
```

- Lex includes y.tab.h and utilizes definitions for token values
- To obtain tokens, yacc calls function yylex() that has a return type of int and returns the token value
- Lex variable yylval returns attributes associated with tokens





NPTEL ONLINE CERTIFICATION COURSES

Thank
you!

yacc Input file

```
%{  
    int yylex(void);  
    void yyerror(char *);  
}  
%token INTEGER  
%%  
program:  
    program expr '\n'      {printf("%d\n", $2);}  
    |  
    ;
```



yacc input file (Contd.)

expr:

```
INTEGER    {$$ = $1;}  
| expr '+' expr      {$$ = $1 + $3;}  
| expr '-' expr      {$$ = $1 - $3;}  
;  
%%  
Void yyerror(char *s) {  
    fprintf(stderr, "%s\n", s);  
    return 0;  
}  
Int main(void) {  
    yyparse();  
    return 0;  
}
```

- yacc can determine shift/reduce and reduce/reduce conflicts.
- shift/reduce resolved in favour of shift
- reduce/reduce conflict resolved in favour of first rule



Syntax Directed Translation

- At the end of parsing, we know if a program is grammatically correct
- Many other things can be done towards code generation by defining a set of semantic actions for various grammar rules
- This is known as Syntax Directed Translation
- A set of attributes associated with grammar symbols
- Actions may be written corresponding to production rules to manipulate these attributes
- Parse tree with attributes is called an annotated parse tree



Example – generate postfix expression

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

$T \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

Attribute val of E and T holds
the string corresponding to the
postfix expression

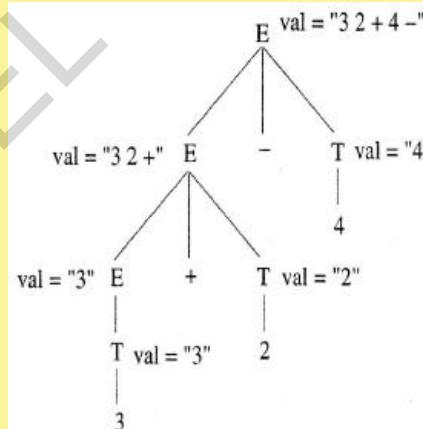
$E \rightarrow E_1 + T \quad \{E.val = E_1.val \mid\mid T.val \mid\mid '+'\}$

$E \rightarrow E_1 - T \quad \{E.val = E_1.val \mid\mid T.val \mid\mid '-'\}$

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

$T \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9 \quad \{T.val = \text{number}\}$

Input string: "3 + 2 - 4"



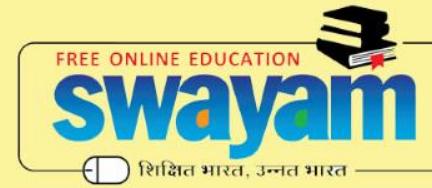
|| means string concatenation



Conclusion

- Seen various types of parsers for syntax analysis
- Error detection and recovery can be integrated with parsers
- Parse tree produced implicitly or explicitly by parsers
- Parse tree can be used in the code generation process





NPTEL ONLINE CERTIFICATION COURSES

Thank
you!



NPTEL ONLINE CERTIFICATION COURSES

Compiler Design Type Checking

Santanu Chattopadhyay
Electronics and Electrical Communication Engineering

CONCEPTS COVERED

- What is a Type Checking
- Static vs. Dynamic Checking
- Type Expressions
- Type Equivalence
- Type Conversion
- Phases of a Compiler
- Conclusion



What is Type Checking

- One of the most important semantic aspects of compilation
- Allows the programmer to limit what types may be used in certain circumstances
- Assigns types to values
- Determines whether these values are used in an appropriate manner
- Simplest situation: check types of objects and report a type-error in case of a violation
- More complex: incorrect types may be corrected (type coercing)



Static vs. Dynamic Checking

- Static Checking
 - Type checking done at compile time
 - Properties can be verified before program run
 - Can catch many common errors
 - Desirable when faster execution is important
- Dynamic Checking
 - Performed during program execution
 - Permits programmers to be less concerned with types
 - Mandatory in some situations, such as, array bounds check
 - More robust and clearer code



Type Expressions

- Used to represent types of language constructs
- A type expression can be
 - Basic type: integer, real, char, Boolean and other atomic types that do not have internal structure. A special type, type-error is used to indicate type violations
 - Type name
 - Type constructor applied to a list of type expressions



Type Expressions

- Arrays are specified as array(I,T), where T is a type and I is an integer or a range of integers. For example, C declaration “int a[100]” identifies type of a to be array(100, integer)
- If T1 and T2 are type expressions, $T1 \times T2$ represents “anonymous records”. For example, an argument list passed to a function with first argument integer and second real, has type integer \times real



Type Expressions

- Named records are products with named elements. For a record structure with two named fields – length (an integer) and word (of type array(10, char)), the record is of type
 $\text{record}((\text{length} \times \text{integer}) \times (\text{word} \times \text{array}(10, \text{character})))$
- If T is a type expression, pointer(T) is also a type expression, representing objects that are pointers to objects of type T
- Function maps a collection of types to another, represented by $D \rightarrow R$, where D is the domain and R is the range of the function.



Type Expressions

- Type expression “integer \times integer \rightarrow character” represents a function that takes two integers as arguments and returns a character value
- Type expression “integer \rightarrow (real \rightarrow character)” represents a function that takes an integer as an argument and returns another function which maps a real number to a character



Type Systems

- Type system of a language is a collection of rules depicting the type expression assignments to program objects
- Usually done with syntax directed definition
- ‘Type checker’ is an implementation of a type system



Strongly Typed Language

- Compiler can verify that the program will execute without any type errors
- All checks are made statically
- Also called a sound type system
- Completely eliminates necessity of dynamic type checking
- Most programming languages are weakly typed
- Strongly typed languages put lot of restrictions
- There are cases in which a type error can be caught dynamically only
- Many languages also allow the user to override the system



Type Checking of Expressions

- Use synthesized attribute ‘type’ for the nonterminal E representing an expression

Expression	Action
$E \rightarrow id$	$E.type \leftarrow lookup(id.entry)$
$E \rightarrow E_1 op E_2$	$E.type \leftarrow \text{if } E_1.type = E_2.type \text{ then } E_1.type \text{ else type-error}$
$E \rightarrow E_1 relop E_2$	$E.type \leftarrow \text{if } E_1.type = E_2.type \text{ then boolean else type-error}$
$E \rightarrow E_1[E_2]$	$E.type \leftarrow \text{if } E_2.type = \text{integer} \text{ and } E_1.type = \text{array}(s,t) \text{ then } t \text{ else type-error}$
$E \rightarrow E_1 \uparrow$	$E.type \leftarrow \text{if } E_1.type = \text{pointer}(t) \text{ then } t \text{ else type-error}$



Type Checking of Statements

- Statements normally do not have any value, hence of type void
- For propagating type error occurring in some statement nested deep inside a block, a set of rules needed

$S \rightarrow id = E$	$S.type \leftarrow$ if $id.type = E.type$ then <i>void</i> else <i>type-error</i>
$S \rightarrow \text{if } E \text{ then } S_1$	$S.type \leftarrow$ if $E.type = \text{boolean}$ then $S_1.type$ else <i>type-error</i>
$S \rightarrow \text{while } E \text{ do } S_1$	$S.type \leftarrow$ if $E.type = \text{boolean}$ then $S_1.type$ else <i>type-error</i>
$S \rightarrow S_1; S_2$	$S.type \leftarrow$ if $S_1.type = \text{void}$ and $S_2.type = \text{void}$ then <i>void</i> else <i>type-error</i>



Type Checking of Functions

- A function call is equivalent to the application of one expression to another

$$\boxed{E \rightarrow E_1(E_2) \mid E.type \leftarrow \text{if } E_2.type = s \text{ and } E_1.type = s \rightarrow t \text{ then } t \text{ else type-error}}$$


Type Equivalence

- It is often needed to check whether two type expressions ‘s’ and ‘t’ are same or not
- Can be answered by deciding equivalence between the two types
- Two categories of equivalence
 - Name equivalence
 - Structural equivalence



Name Equivalence

- Two types are name equivalent if they have same name or label

```
typedef int Value
```

```
typedef int Total
```

```
...
```

```
Value var1, var2
```

```
Total var3, var4
```

- Variables var1, var2 are name equivalent, so are var3 and var4
- Variables var1 and var4 are not name equivalent, as their type names are different



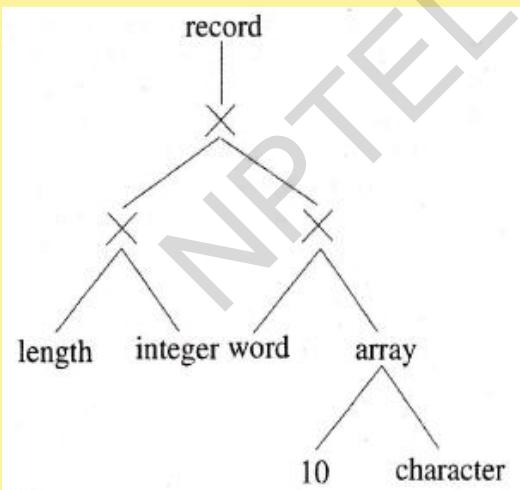
Structural Equivalence

- Checks the structure of the type
- Determines equivalence by checking whether they have same constructor applied to structurally equivalent types
- Checked recursively
- Types array(I1, T1) and array(I2, T2) are structurally equivalent if I1 and I2 are equal and T1 and T2 are structurally equivalent



Directed Acyclic Graph Representation

- Type expressions can be represented as a DAG or a tree
- “record((length × integer) × (word × array(10, character)))”



Function dag_equivalence

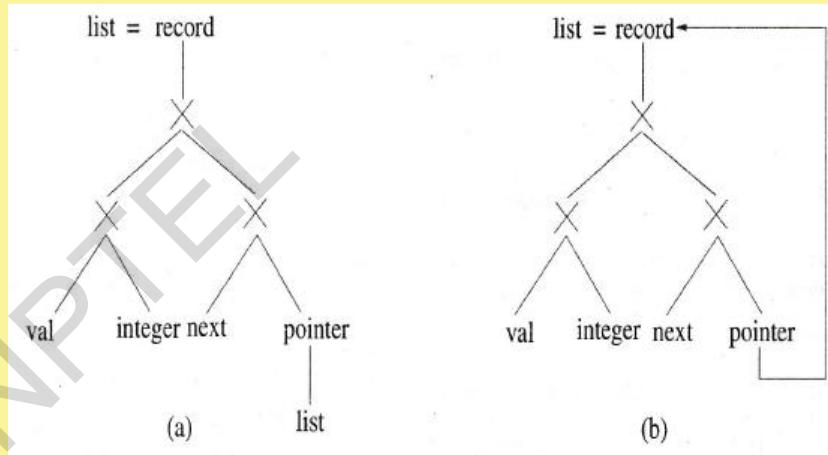
```
function dag-equivalence(s,t: type-DAGs): boolean
begin
    if s and t represents the same basic type then return true
    if s represents array(I1, T1) and t represents array(I2, T2) then
        if I1 = I2 then return dag-equivalence(T1, T2)
        else return false
    if s represents s1 × s2 and t represents t1 × t2 then
        return dag-equivalence(s1, t1) and dag-equivalence(s2, t2)
    if s represents pointer(s1) and t represents pointer(t1) then
        return dag-equivalence(s1, t1)
    if s = s1 → s2 and t = t1 → t2 then
        return dag-equivalence(s1, t1) and dag-equivalence(s2, t2)
    return false
end.
```



Cycles in Type Representation

- Some languages allow types to be defined in a cyclical fashion

```
struct list  
{  
    int val;  
    struct list *next;  
}
```



- (a) Acyclic representation (b) Cyclic representation



Cycles in Type Representation

- Most programming languages, including C, uses acyclic one
- Type names are to be declared before using it, excepting pointers
- Name of the structure is also part of the type
- Equivalence test stops when a structure is reached
- At this point, type expressions are equivalent if they point to the same structure name, nonequivalent otherwise



Type Conversion

- Refers to local modification of type for a variable or subexpression
- For example, it may be necessary to add an integer quantity to a real variable, however, the language may require both the operands to be of same type
- Modifying integer variable to real will require more space
- Solution: to treat integer operand as really operand locally and perform the operation
- May be done explicitly or implicitly
- Implicit conversion → type coercion

```
int x;  
float y;  
...  
y = ((float)x)/14.0
```

```
int x;  
float y;  
...  
y = x/14.0
```



Conclusion

- Compilers usually perform static type checking
- Dynamic type checking is costly
- Types are normally represented as type expressions
- Type checking can be performed by syntax directed techniques
- Type graphs may be compared to check type equivalence





NPTEL ONLINE CERTIFICATION COURSES

Thank
you!

Compiler Design

Symbol Tables

Santanu Chattopadhyay

Electronics and Electrical Communication Engineering



- Information in Symbol Table
- Features of Symbol Table
- Simple Symbol Table
- Scoped Symbol Table
- Conclusion



Introduction

- Essential data structure used by compilers to remember information about identifiers in the source program
- Usually lexical analyzer and parser fill up the entries in the table, later phases like code generator and optimizer make use of table information
- Types of symbols stored in the symbol table include variables, procedures, functions, defined constants, labels, structures etc.
- Symbol tables may vary widely from implementation to implementation, even for the same language



Information in Symbol Table

- Name
 - Name of the identifier
 - May be stored directly or as a pointer to another character string in an associated string table – names can be arbitrarily long
- Type
 - Type of the identifier: variable, label, procedure name etc.
 - For variables, its type: basic types, derived types etc.
- Location
 - Offset within the program where the identifier is defined
- Scope
 - Region of the program where the current definition is valid
- Other attributes: array limits, fields of records, parameters, return values etc.



Usage of Symbol Table Information

- Semantic Analysis – check correct semantic usage of language constructs, e.g. types of identifiers
- Code Generation – Types of variables provide their sizes during code generation
- Error Detection – Undefined variables. Recurrence of error messages can be avoided by marking the variable type as undefined in the symbol table
- Optimization – Two or more temporaries can be merged if their types are same



Operations on Symbol Table

- Lookup – Most frequent, whenever an identifier is seen it is needed to check its type, or create a new entry
- Insert – Adding new names to the table, happens mostly in lexical and syntax analysis phases
- Modify – When a name is defined, all information may not be available, may be updated later
- Delete – Not very frequent. Needed sometimes, such as when a procedure body ends



Issues in Symbol Table Design

- Format of entries – Various formats from linear array to tree structured table
- Access methodology – Linear search, Binary search, Tree search, Hashing, etc.
- Location of storage – Primary memory, partial storage in secondary memory
- Scope Issues – In block-structured language, a variable defined in upper blocks must be visible to inner blocks, not the other way



Simple Symbol Table

- Works well for languages with a single scope
- Commonly used techniques are
 - Linear table
 - Ordered list
 - Tree
 - Hash table



Linear Table

- Simple array of records with each record corresponding to an identifier in the program
- Example:

```
int x, y  
real z  
...  
procedure abc  
...  
L1:  
...
```

Name	Type	Location
x	integer	Offset of x
y	integer	Offset of y
z	real	Offset of z
abc	procedure	Offset of abc
L1	label	Offset of L1



Linear Table

- If there is no restriction in the length of the string for the name of an identifier, string table may be used, with name field holding pointers
- Lookup, insert, modify take $O(n)$ time
- Insertion can be made $O(1)$ by remembering the pointer to the next free index
- Scanning most recent entries first may probably speed up the access – due to program locality – a variable defined just inside a block is expected to be referred to more often than some earlier variables



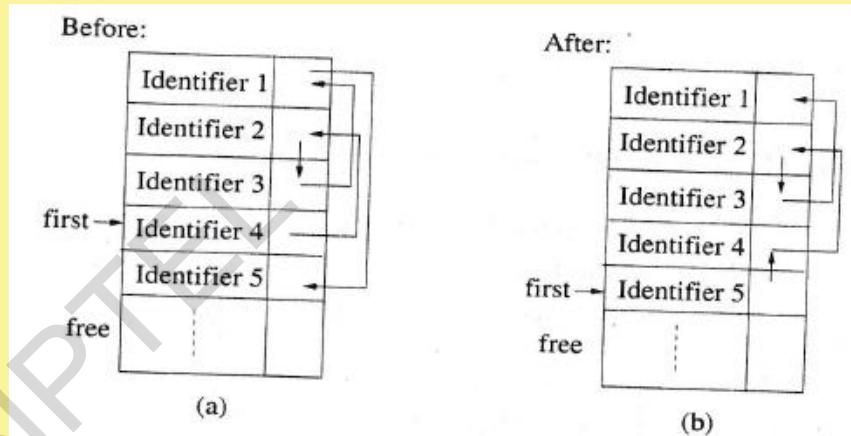
Ordered List

- Variation of linear tables in which list organization is used
- List is sorted in some fashion , then binary search can be used with $O(\log n)$ time
- Insertion needs more time
- A variant – self-organizing list: neighbourhood of entries changed dynamically



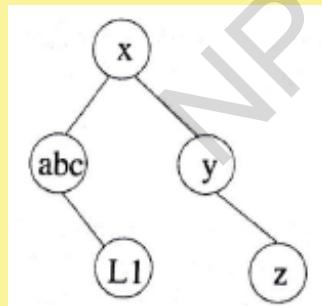
Self-Organizing List

- In Fig (a), Identifier4 is the most recently used symbol, followed by Identifier2, Identifier3 and so on
- In Fig (b), Identifier5 is accessed next, accordingly the order changes
- Due to program locality, it is expected that during compilation, entries near the beginning of the ordered list will be accessed more frequently
- This improves lookup time



Tree

- Each entry represented by a node of the tree
- Based on string comparison of names, entries lesser than a reference node are kept in its left subtree, otherwise in the right subtree
- Average lookup time $O(\log n)$
- Proper height balancing techniques need to be utilized



Hash Table

- Useful to minimize access time
- Most common method for implementing symbol tables in compilers
- Mapping done using Hash function that results in unique location in the table organized as array
- Access time $O(1)$
- Imperfection of hash function results in several symbols mapped to the same location – collision resolution strategy needed
- To keep collisions reasonable, hash table is chosen to be of size between n and $2n$ for n keys



Desirable Properties of Hash Functions

- Should depend on the name of the symbol. Equal emphasis be given to each part
- Should be quickly computable
- Should be uniform in mapping names to different parts of the table. Similar names (such as, data1 and data2) should not cluster to the same address
- Computed value must be within the range of table index



Scoped Symbol Table

- Scope of a symbol defines the region of the program in which a particular definition of the symbol is valid – definition is *visible*
- Block structured languages permit different types of scopes for the identifiers – *scope rules* for the language
 - Global scope: visibility throughout the program, global variables
 - File-wide scope: visible only within the file
 - Local scope within a procedure: visible only to the points inside the procedure, local variables
 - Local scope within a block: visible only within the block in which it is defined



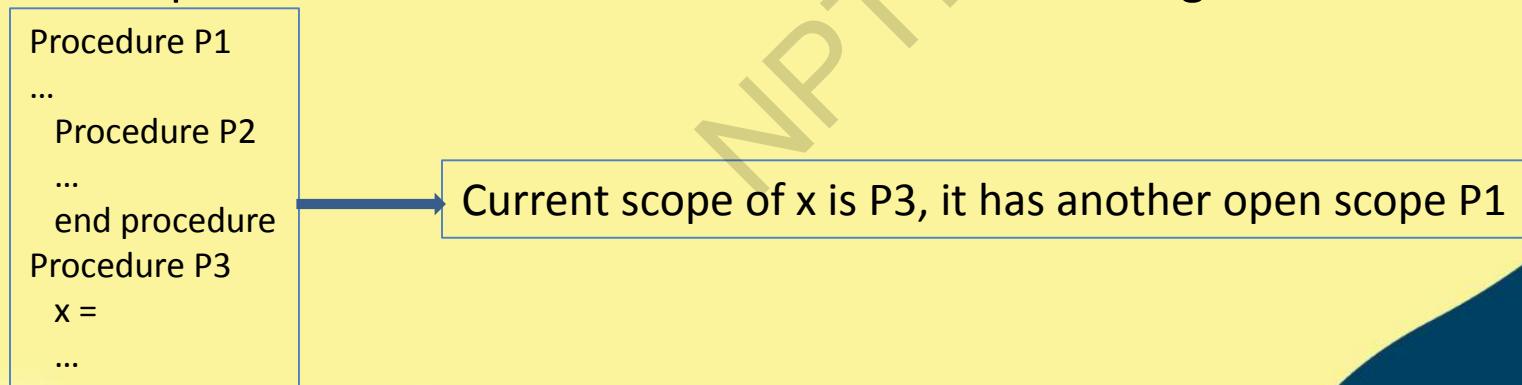
Scoping Rules

- Two categories depending on the time at which the scope gets defined
- Static or Lexical Scoping
 - Scope defined by syntactic nesting
 - Can be used efficiently by the compiler to generate correct references
- Dynamic or Runtime Scoping
 - Scoping depends on execution sequence of the program
 - Lot of extra code needed to dynamically decide the definition to be used



Nested Lexical Scoping

- To reach the definition of a symbol, apart from the current block, the blocks that contain this innermost one, also have to be considered
- Current scope is the innermost one
- There exists a number of open scopes – one corresponding to the current scope and others to each of the blocks surrounding it



Visibility Rules

- Used to resolve conflicts arising out of same variable being defined more than once
- If a name is defined in more than one scope, the innermost declaration closest to the reference is used to interpret
- When a scope is exited all declared variables in that scope are deleted and the scope is thus *closed*
- Two methods to implement symbol tables with nested scope
 - One table for each scope
 - A single global table

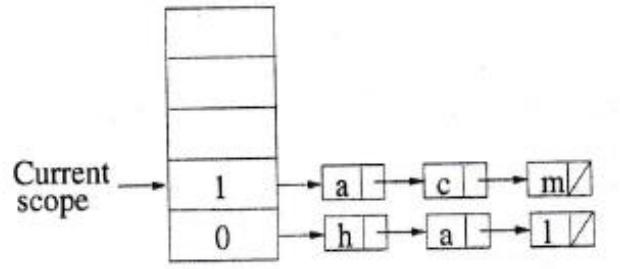


One Table Per Scope

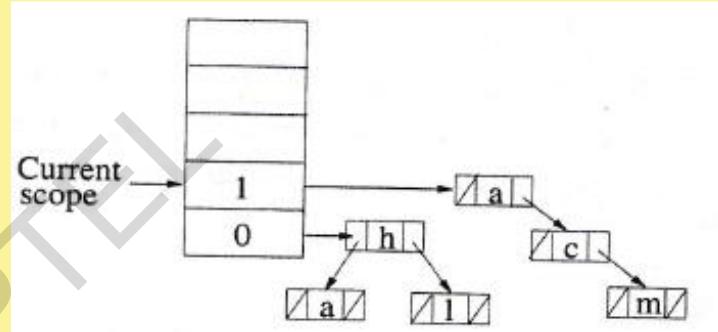
- Maintain a different table for each scope
- A stack is used to remember the scopes of the symbol tables
- Drawbacks:
 - For a single-pass compiler, table can be popped out and destroyed when a scope is closed, not for a multi-pass compiler
 - Search may be expensive if variable is defined much above in the hierarchy
 - Table size allotted to each block is another issue
- Lists, Trees, Hash Tables can be used



One Table Per Scope



Scoped Symbol Table – Lists



Scoped Symbol Table – Trees

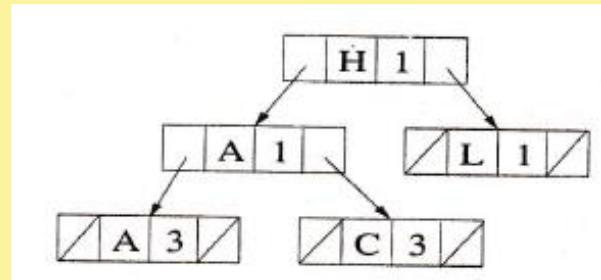
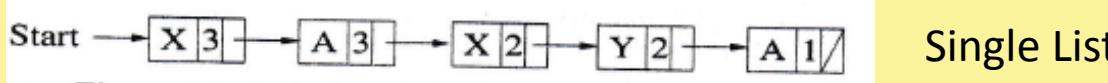


One Table for All Scopes

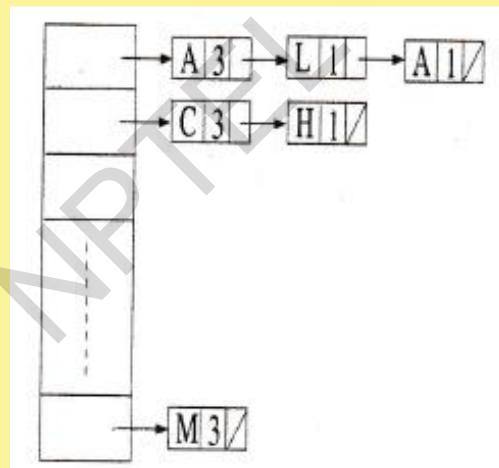
- All identifiers are stored in a single table
- Each entry in the symbol table has an extra field identifying the scope
- To search for an identifier, start with the highest scope number, then try out the entries having next lesser scope number, and so on
- When a scope gets closed, all identifiers with that scope number are removed from the table
- Suitable particularly for single-pass compilers
- List, Tree and Hash Table can be used



One Table for All Scopes



Tree



Hash Table



Conclusions

- Symbol table, though not part of code generated by the compiler, helps in the compilation process
- Phases like Lexical Analysis and Syntax Analysis produce the symbol table, while other phases use its content
- Depending upon the scope rules of the language, symbol table needs to be organized in various different manners
- Data structures commonly used for symbol table are linear table, ordered list, tree, hash table, etc.



Thank you

NPTEL



Compiler Design

Runtime Environment Management

Santanu Chattopadhyay

Electronics and Electrical Communication Engineering



- What is Runtime Environment
- Activation Record
- Environment without Local Procedures
- Environment with Local Procedures
- Display
- Conclusion



What is Runtime Environment

- Refers to the program snap-shot during execution
- Three main segments of a program
 - Code
 - Static and global variables
 - Local variables and arguments
- Memory needed for each of these entities
 - Generated code: Text for procedures and programs. Size known at compile time. Space can be allotted statically before execution
 - Data objects:
 - Global variables/constants – space known at compile time
 - Local variables – space known at compile time
 - Dynamically created variables – space (heap) in response to memory allocation requests
 - Stack: To keep track of procedure activations



Logical Address Space of Program



- Code occupies the lowest portion
- Global variables are allocated in the static portion
- Remaining portion of the address space, stack and heap are allocated from the opposite ends to have maximum flexibility



Activation Record

- Storage space needed for variables associated with each activation of a procedure – *activation record or frame*
- Typical activation record contains
 - Parameters passed to the procedure
 - Bookkeeping information, including return values
 - Space for local variables
 - Space for compiler generated local variables to hold sub-expression values



Location for Activation Record

- Depending upon language, activation record can be created in the static, stack or heap area
- Creation in Static Area:
 - Early languages, like FORTRAN
 - Address of all arguments, local variables etc. are preset at compile time itself
 - To pass parameters, values are copied into these locations at the time of invoking the procedure and copied back on return
 - There can be a single activation of a procedure at a time
 - Recursive procedures cannot be implemented



Location for Activation Record

- Creation in Stack Area:
 - Used for languages like C, Pascal, Java etc.
 - As and when a procedure is invoked, corresponding activation record is pushed onto the stack
 - On return, entry is popped out
 - Works well if local variables are not needed beyond the procedure body
- For languages like LISP, in which a full function may be returned, activation record created in the heap



Processor Registers

- Also a part of the runtime environment
- Used to store temporaries, local variables, global variables and some special information
- Program counter points to the statement to be executed next
- Stack pointer points to the top of the stack
- Frame pointer points to the current activation record
- Argument pointer points to the area of the activation record reserved for arguments



Environment Types

- Stack based environment without local procedures – common for languages like C
- Stack based environment with local procedures – followed for block structured languages like Pascal



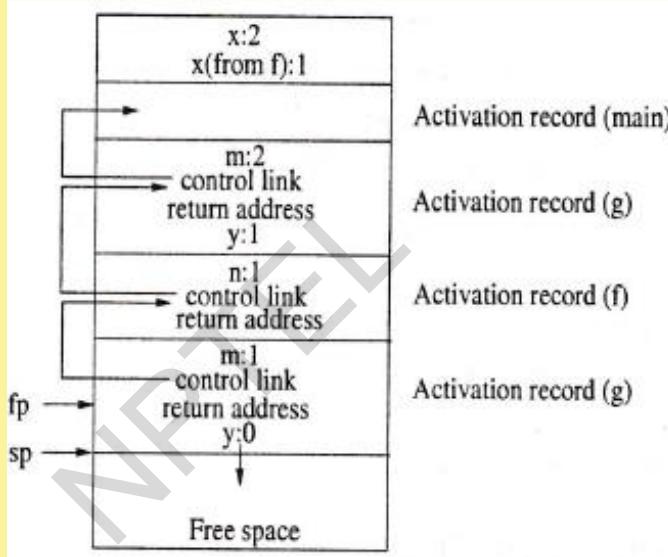
Environment without Local Procedures

- For languages where all procedures are global
- Stack based environment needs two things about activation records
 - Frame pointer: Pointer to the current activation record to allow access to local variables and parameters
 - Control link / Dynamic link: Kept in current activation record to record position of the immediately preceding activation record



Environment without Local Procedures

```
int x = 2;  
void f( int n ) {  
    static int x = 1;  
    g(n);  
    x--;  
}  
void g( int m ) {  
    int y = m - 1;  
    if (y > 0) {  
        f(y);  
        x--;  
    }  
}  
main() {  
    g(x); return 0;  
}
```

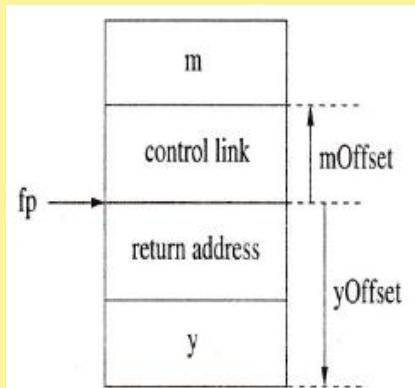


Snapshot of program execution after main has called g, g has called f and f has in turn called g



Accessing Variables

- Parameters and local variables found by offset from the current frame pointer
- Offsets can be calculated statically by the compiler
- Consider procedure g with parameter m and local variable y



$m\text{Offset} = \text{size of control link} = +4 \text{ bytes}$
 $y\text{Offset} = -(\text{size of } y + \text{size of return address}) = -6$
Hence, m and y can be accessed by $4(fp)$ and $-6(fp)$



Activation Record Creation

At a call	
Caller	Callee
1. Allocate basic frame 2. Store parameters 3. Store return address 4. Save caller-saved registers 5. Store self frame pointer 6. Set frame pointer for child 7. Jump to child	1. Save callee saved registers, state 2. Extend frame for locals 3. Initialize locals 4. Fall through to code
At a return	
Caller	Callee
1. Copy return value 2. Deallocate basic frame 3. Restore caller-saved registers	1. Store return value 2. Restore callee-saved registers, state 3. Unextend frame 4. Restore parent's frame pointer 5. Jump to return address



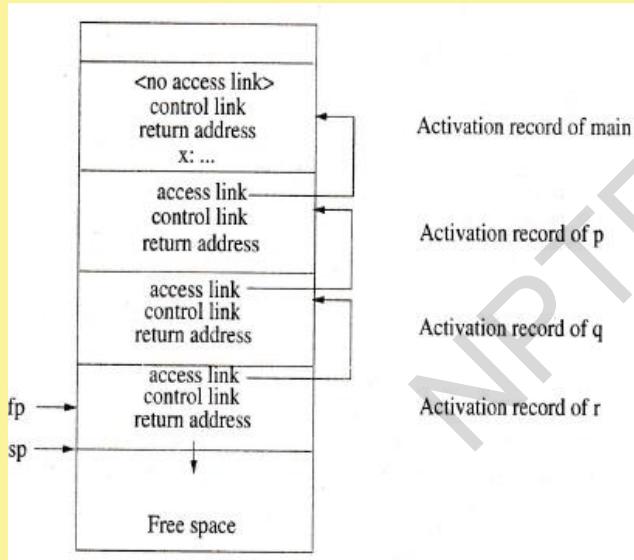
Environment with Local Procedures

- For supporting local procedures, variables may have various scopes
- To determine the definition to be used for a reference to a variable, it is needed to access non-local, non-global variables
- These definitions are local to one of the procedures nesting the current one – need to look into the activation records of nesting procedures
- Solution is to keep extra bookkeeping information, called *access link*, pointing to the activation record for the defining environment of a procedure



Environment with Local Procedures

```
program chaining;  
procedure p;  
var x: integer;  
procedure q;  
procedure r  
begin  
    x := 2;  
    ...  
    if ... Then p;  
end {of r}  
begin  
    r;  
end {of q}  
begin  
    q;  
end {of p}  
begin {of main}  
    p;  
end.
```



- Current procedure r
- To locate definition of x, it has to traverse through the activation records using access links
- When the required procedure containing definition of x is reached, it is accessed via offset from the corresponding frame pointer



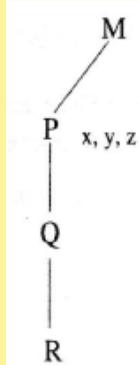
Compiler's Responsibility

- Proper code to access the correct definitions:
 - Find difference d between the lexical nesting level of declaration of the name and the lexical nesting level of the procedure referring to it
 - Generate code for following d access links to reach the right activation record
 - Generate code to access the variable through offset mechanism

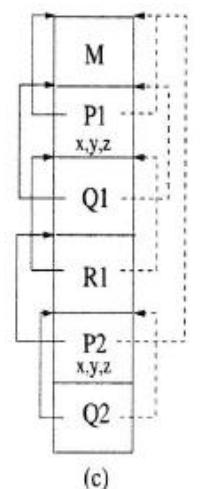
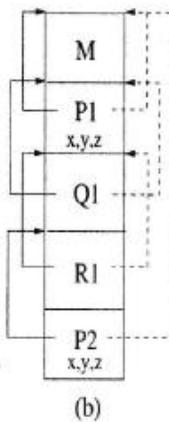
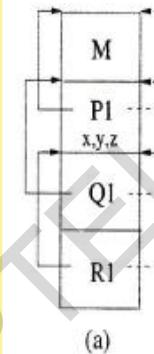


Example

```
program M;  
procedure P;  
  var x, y, z;  
procedure Q;  
procedure R;  
begin  
  ... z = P; ...  
end R;  
begin  
  ... y = R; ...  
end Q;  
begin  
  ... x = Q; ...  
end P;  
begin  
  ... P; ...  
end M;
```



Lexical level(M) = 1
Lexical level(P) = 2
Lexical level(Q) = 3
Lexical level(R) = 4



DISPLAY

- Difficulty in non-local definitions is to search by following access links
- Particularly for virtual paging environment, certain portion of the stack containing activation records may be swapped out, access may be very slow
- To access variables without search, *display* is used



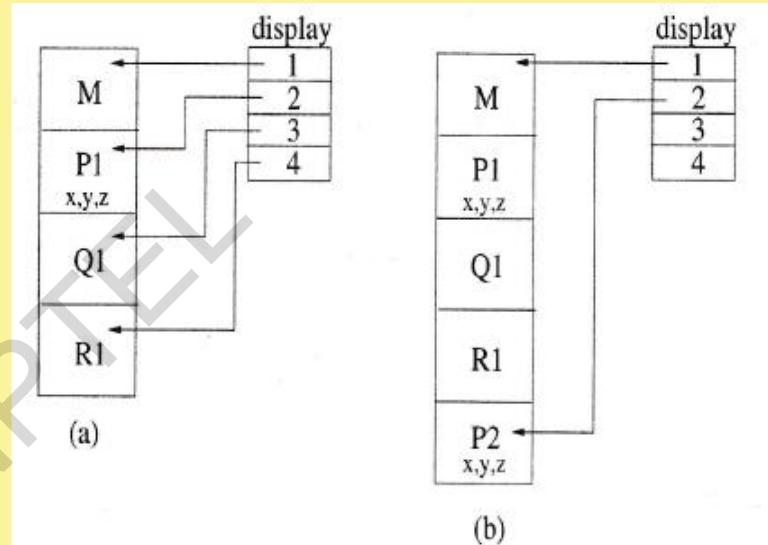
Display

- Display d is a global array of pointers to activation records, indexed by the lexical nesting depth
- Element $d[i]$ points to the most recent activation of the block at nesting depth i
- A nonlocal X is found as follows:
 - If the most closely nested declaration of X is at nesting depth i , then $d[i]$ points to activation record containing the location for X
 - Use relative address within the activation record to access X



Example

- Maximum nesting depth 4, so 4 entries in the display
- In Fig (a), M has called P, P has called Q and Q has in turn called R
- Compiler knows that x is in procedure P at lexical level 2
- Code is generated to access second entry of the display to reach the activation record of P directly
- Same is in Fig (b)



Maintaining Display

- When a procedure P at nesting depth i is called, following actions are taken:
 - Save value of $d[i]$ in the activation record for P
 - Set $d[i]$ to point to new activation record
- When a procedure P finishes:
 - $d[i]$ is reset to the value stored in the activation record of P



Example

```
1. Program X  
2. var x, y, z;  
3. Procedure P  
4.     var a;  
5. begin (of P)  
6.     a = Q  
7. end (of P)  
8. Procedure Q  
9.     Procedure R  
10.    begin (of R)  
11.        P  
12.    end (of R)  
13.    begin (of Q)  
14.        R  
15.    end (of Q)  
16.    begin (of X)  
17.        P  
18.        Q  
19.    end (of X)
```

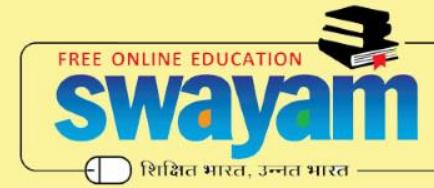
- Show the snapshots of the stack of activation records at the time of executing line nos.
6, 11, 14, 17, 18
- Show the corresponding displays



Conclusion

- Data structure activation record contains necessary information to control program execution
- Compiler writer must generate appropriate code for operations
- A small array, display, helps in the process
- Display management becomes a part of compiler's responsibility





NPTEL ONLINE CERTIFICATION COURSES

Thank
you!

Compiler Design

Intermediate Code Generation

Santanu Chattopadhyay

Electronics and Electrical Communication Engineering



- Intermediate Languages
- Intermediate Language Design Issues
- Intermediate Representation Techniques
- Statements in Three-Address Code
- Implementation of Three-Address Instructions
- Three-Address Code Generation
- Conclusion



Intermediate Code

- Compilers are designed to produce a representation of input program in some hypothetical language or data structure
- Representations between the source language and the target machine language programs
- Offers several advantages
 - Closer to target machine, hence easier to generate code from
 - More or less machine independent, makes it easier to retarget the compiler to various different target processors
 - Allows variety of machine-independent optimizations
 - Can be implemented via syntax-directed translation, can be folded into parsing by augmenting the parser



Intermediate Languages

- Can be classified into – High-level representation and Low-level representation

High-level Representation

- Closer to source language program
- Easy to generate from input program
- Code optimization difficult, since input program is not broken down sufficiently

Low-level Representation

- Closer to target machine
- Easy to generate final code from
- Good amount of effort in generation from the source code



Intermediate Language Design Issues

- Set of operators in intermediate language must be rich enough to allow the source language to be implemented
- A small set of operations in the intermediate language makes it easy to retarget
- Intermediate code operations that are closely tied to a particular machine or architecture can make it harder to port
- A small set of intermediate code operations may lead to long instruction sequences for some source language constructs. Implies more work during optimization



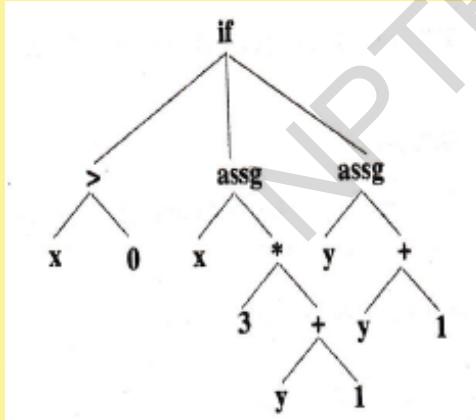
Intermediate Representation Techniques

- High-level Representation
 - Abstract Syntax Trees
 - Directed Acyclic Graphs
 - P-code
- Low-level Representation



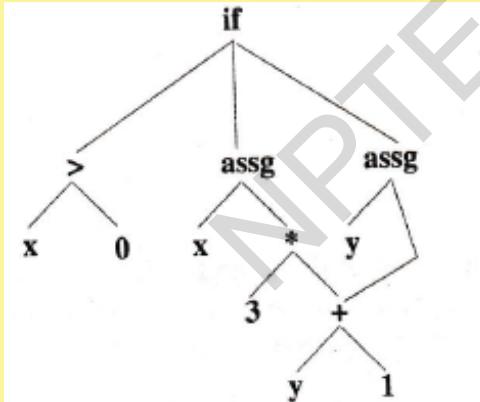
Abstract Syntax Tree

- Compact form of parse tree
- Represents hierarchical structure of a program
- Nodes represent operators, children of a node the operands
- Example: “if $x > 0$ then $x = 3 * (y + 1)$ else $y = y + 1$ ”



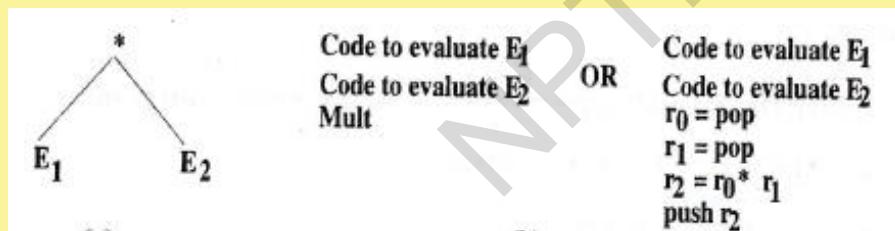
Directed Acyclic Graph (DAG)

- Similar to syntax tree
- Common subexpressions represented by single node



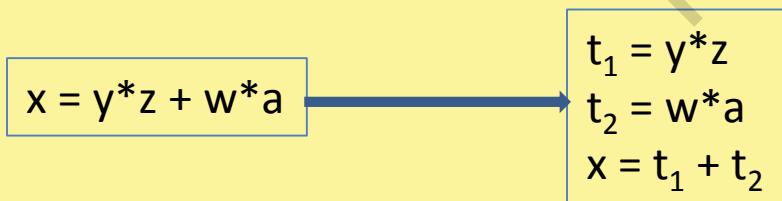
P-code

- Used for stack based virtual machines
- Operands are always found on the top of the stack
- May need to push operands to the stack first
- Syntax tree to P-code:



Low-level Representation – Three Address Code

- Sequence of instructions of the form “ $x = y \text{ op } z$ ”
- Only one operator permitted in the right hand side
- Due to its simplicity, offers better flexibility in terms of target code generation and code optimization



Statements in Three-Address Code

- Intermediate languages usually have the following types of statements
 - Assignment
 - Jumps
 - Address and Pointer Assignments
 - Procedure Call/Return
 - Miscellaneous



Assignment Statement

- Three types of assignment statements
 - $x = y \text{ op } z$, op being a binary operator
 - $x = \text{op } y$, op being a unary operator
 - $x = y$
- For all operators in the source language, there should be a counterpart in the intermediate language



Jump Statement

- Both conditional and unconditional jumps are required
 - goto L, L being a label
 - if x relop y goto L



Indexed Assignment

- Only one-dimensional arrays need to be supported
- Arrays of higher dimensions are converted to one-dimensional arrays
- Statements to be supported
 - $x = y[i]$
 - $x[i] = y$



Address and Pointer Assignments

- Statements required are of following types
 - $x = \&y$, address of y assigned to x
 - $x = *y$, content of location pointed to by y is assigned to x
 - $x = y$, simple pointer assignment, where x and y are pointer variables



Procedure Call/Return

- A call to the procedure $P(x_1, x_2, \dots, x_n)$ is converted as

param x_1

param x_2

...

param x_n

- A procedure is implemented using the following statements

enter f , Setup and initialization

leave f , Cleanup actions (if any)

return

return x

retrieve x , Save returned value in x



Miscellaneous Statements

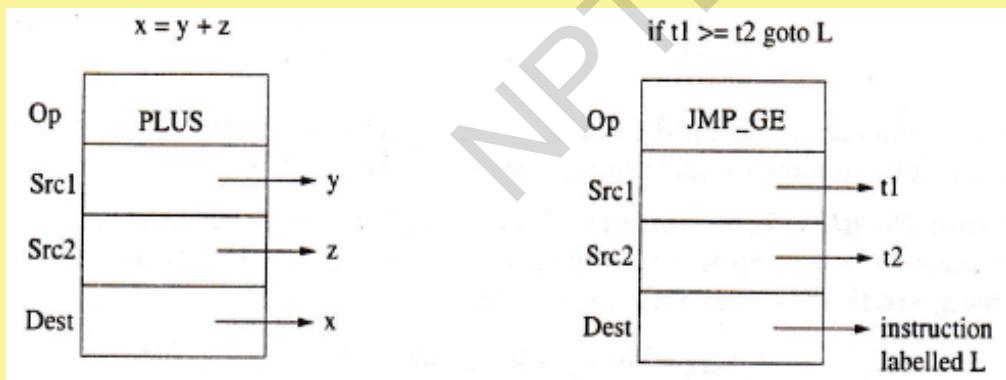
- More statements may be needed depending upon the source language
- One such statement is to define jump target as,

label L



Three-Address Instruction Implementation

- Quadruple representation – each instruction has at most four fields:
 - Operation – identifying the operation to be carried out
 - Up to two operands – a bit is used to indicate whether it is a constant or a pointer
 - Destination



Example

```
if x+2 > 3*(y-1)+4 then z = 0
```

```
t1 = x + 2  
t2 = y - 1  
t3 = 3 * t2  
t4 = t3 + 4  
if t1 ≤ t4 goto L  
z = 0  
Label L
```



Three Address Code Generation - Assignment

Grammar:

$$S \rightarrow id := E$$

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

Attributes for non-terminal E:

- E.place – name that will hold value of E
- E.code – sequence of three address statements corresponding to evaluation of E

Attributes for non-terminal S:

- S.code – sequence of three-address statements

Attributes for terminal symbol id:

- id.place – contains the name of the variable to be assigned

Grammar Rule	Semantic Actions
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp();$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place + E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp();$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place * E_2.place)$
$E \rightarrow -E_1$	$E.place := newtemp();$ $E.code := E_1.code \parallel gen(E.place := 'uminus' E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place;$ $E.code := ''$

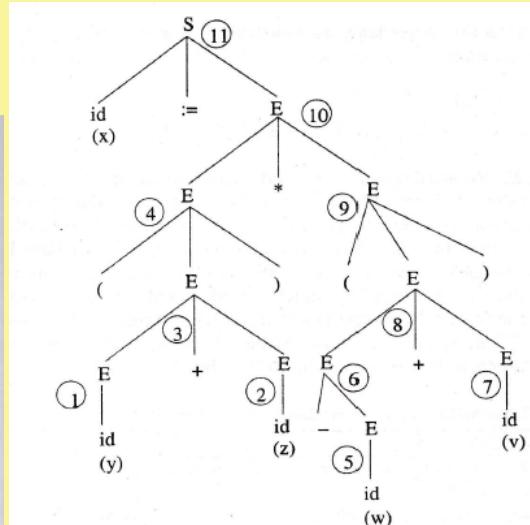
- Function *newtemp* returns a unique new temporary variable.
- Function *gen* accepts a string and produces it as a three-address quadruple.
- ‘|’ concatenates two three-address code segments



Example

$x := (y+z)^*(-w+v)$

Reduction No.	Action
1	$E.place = y$
2	$E.place = z$
3	$E.place = t_1$
	$E.code = \{t_1 := y + z\}$
4	$E.place = t_1$
	$E.code = \{t_1 := y + z\}$
5	$E.place = w$
6	$E.place = t_2$
	$E.code = \{t_2 := uminus w\}$
7	$E.place = v$
8	$E.place = t_3$
	$E.code = \{t_2 := uminus w, t_3 := t_2 + v\}$
9	$E.place = t_3$
	$E.code = \{t_2 := uminus w, t_3 := t_2 + v\}$
10	$E.place = t_4$
	$E.code = \{t_1 := y + z, t_2 := uminus w, t_3 := t_2 + v, t_4 := t_1 * t_3\}$
11	$S.code = \{t_1 := y + z, t_2 := uminus w, t_3 := t_2 + v, t_4 := t_1 * t_3, x := t_4\}$



Code Generation for Arrays

- Consider array element $A[i]$
- Assume lowest and highest indices of A are low and $high$, width of each element w and start address of A , $base$
- Element $A[i]$ starts at location $(base + (i - low)*w) = ((base - low*w) + i*w)$
- First part of the expression can be precomputed into a constant and added to the offset $i*w$



Code Generation for Arrays

- For two-dimensional array with row-major storage,
 $A[i_1, i_2]$ starts at location

$$\text{base} + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w$$

$$= \text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w + ((i_1 * n_2) + i_2) * w$$

where n_2 is the size of the second dimension

- This can be extended to higher dimensions



Array Translation Scheme

Grammar:

$S \rightarrow L := E$

$E \rightarrow E + E \mid (E) \mid L$

$L \rightarrow Elist] \mid id$

$Elist \rightarrow Elist, E \mid id[E]$

Attributes:

- $L.place$: holds name of the variable (may be array name also)
- $L.offset$: null for simple variable, offset of the element for array
- $E.place$: name of the variable holding value of expression E
- $Elist.array$: holds the name of the array referred to
- $Elist.place$: name of the variable holding value for index expression
- $Elist.dim$: holds current dimension under consideration for array



Semantic Actions for Arrays

$S \rightarrow L := E$

{ if L.offset = null then
 emit(L.place ':=' E.place);
else
 emit(L.place '[' L.offset ']' ':=' E.place)
}

$E \rightarrow E1 + E2$

{ E.place := newtemp();
 emit(E.place ':=' E1.place '+' E2.place)
}

$E \rightarrow (E1)$

{ E.place := E1.place }



Semantic Actions for Arrays

$E \rightarrow L$

```
{ if L.offset = null then  
    E.place = L.place  
else  
    E.place = newtemp()  
    emit(E.place ':=' L.place '[' L.offset ']')  
}
```

$L \rightarrow id$

```
{ L.place = id.place  
L.offset ':=' null  
}
```

$L \rightarrow Elist]$

```
{ L.place = newtemp()  
L.offset = newtemp()  
emit(L.place ':=' c(Elist.array) /* c returns constant part of the array */  
emit(L.offset ':=' Elist.place * width(Elist.array))  
}
```



Semantic Actions for Arrays

Elist → Elist1, E

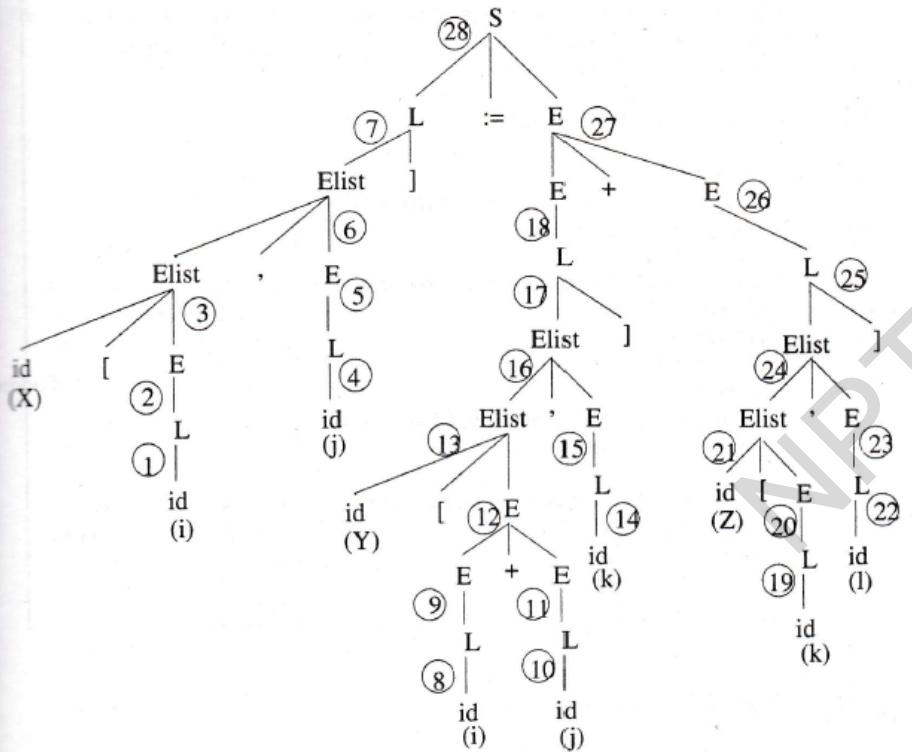
```
{ t = newtemp()
  m = Elist1.dim + 1
  emit(t ':=' Elist1.place '*' limit(Elist1.array, m))
  emit(t ':=' t +' E.place
  Elist.array = Elist1.array
  Elist.place = t
  Elist.dim = m
}
```

Elist → id [E

```
{ Elist.array = id.place
  Elist.place = E.place
  Elist.dim = 1
}
```



Example



$$X[i, j] := Y[i + j, k] + Z[k, l]$$

Array dimensions:

$$X[d_1, d_2], Y[d_3, d_4], Z[d_5, d_6]$$

Each element of width w



Example

Step No.	Attribute assignment	Code generated
1	$L.place = i, L.offset = null$	
2	$E.place = i$	
3	$Elist.array = X, Elist.place = i, Elist.dim = 1$	
4	$L.place = j, L.offset = null$	
5	$E.place = j$	
6	$Elist.array = X, Elist.place = t_1, Elist.dim = 2$	$t_1 = i * d_2, t_1 := t_1 + j$
7	$L.place = t_2, L.offset = t_3$	$t_2 := C(X), t_3 := t_1 * w$
8	$L.place = i, L.offset = null$	
9	$E.place = i$	
10	$L.place = j, L.offset = null$	
11	$E.place = j$	
12	$E.place = t_4$	$t_4 := i + j$
13	$Elist.array = Y, Elist.place = t_4, Elist.dim = 1$	
14	$L.place = k, L.offset = null$	
15	$E.place = k$	
16	$Elist.array = Y, Elist.place = t_5, Elist.dim = 2$	$t_5 := t_4 * d_4, t_5 := t_5 + k$
17	$L.place = t_6, L.offset = t_7$	$t_6 := C(Y), t_7 := t_5 * w$
18	$E.place = t_8$	$t_8 := t_6[t_7]$
19	$L.place = k, L.offset = null$	
20	$E.place = k$	
21	$Elist.array = Z, Elist.place = k, Elist.dim = 1$	
22	$L.place = l, L.offset = null$	
23	$E.place = l$	
24	$Elist.array = Z, Elist.place = t_9, Elist.dim = 2$	$t_9 := k * d_6, t_9 := t_9 + l$
25	$L.place = t_{10}, L.offset = t_{11}$	$t_{10} := C(Z), t_{11} := t_9 * w$
26	$E.place = t_{12}$	$t_{12} := t_{10}[t_{11}]$
27	$E.place = t_{13}$	$t_{13} := t_8 + t_{12}$
28		$t_2[t_3] := t_{13}$



Translation of Boolean Expressions

Attributes of Boolean expression B :

1. $B.\text{true}$: defines place, control should reach if B is true
2. $B.\text{false}$: defines place, control should reach if B is false

Grammar:

$$\begin{array}{l} B \rightarrow B \text{ or } B \\ | \quad B \text{ and } B \\ | \quad \text{not } B \\ | \quad (B) \\ | \quad \text{id relop id} \\ | \quad \text{true} \\ | \quad \text{false} \end{array}$$

- Assumed true and false transfer points for entire expression
- B is known
- If B_1 is true, B is true → need not evaluate B_2 → called short-circuit evaluation
- If B_1 is false, B_2 needs to be evaluated
- Thus, $B_1.\text{false}$ assigned a new label marking beginning of evaluation of B_2
- Function `newlabel()` generates new label

$$\begin{aligned} B &\rightarrow B_1 \text{ or } B_2 \\ &\{ \quad B_1.\text{true} = B.\text{true} \\ &\quad B_1.\text{false} = \text{newlabel}() \\ &\quad B_2.\text{true} = B.\text{true} \\ &\quad B_2.\text{false} = B.\text{false} \\ &\quad B.\text{code} = B_1.\text{code} \parallel \\ &\quad \text{gen}(B_1.\text{false}, ':') \parallel \\ &\quad B_2.\text{code} \\ &\} \end{aligned}$$


Translation of Boolean Expressions

$B \rightarrow B1 \text{ and } B2$

```
{ B1.true = newlabel()  
B1.false = B.false  
B2.true = B.true  
B2.false = B.false  
B.code = B1.code ||  
gen(B1.true, ':') ||  
B2.code  
}
```

$B \rightarrow \text{not } B1$

```
{ B1.true = B.false  
B1.false = B.true  
B.code = B1.code  
}
```

$B \rightarrow (B1)$

```
{ B1.true = B.true  
B1.false = B.false  
B.code = B1.code  
}
```

$B \rightarrow \text{true}$

```
{ B.code = gen( 'goto' B.true) }
```

$B \rightarrow \text{id1 relop id2}$

```
{ B.code = gen('if' id1.place relop id2.place 'goto' B.true) || gen('goto' B.false)}
```

$B \rightarrow \text{false}$

```
{ B.code = gen( 'goto' B.false) }
```



Disadvantages

- Makes the scheme inherently two-pass procedure
- All jump targets are computed in the first pass
- Actual code generation done in second pass
- A single-pass approach can be developed by
 - Modifying the grammar a bit, and
 - Introducing a few more attributes
 - A few new procedures
 - Generated code can be visualized as an array of quadruples



Attributes

- *B.truelist*
 - List of locations within the generated code for *B*, at which *B* definitely true
 - Once defined, all these points should transfer control to *B.true*
- *B.falselist*
 - List of locations within the generated code for *B*, at which *B* definitely false
 - Once defined, all these points should transfer control to *B.false*



Extra Functions

- *makelist(*i*):* creates a new list with a single entry *i* – an index into the array of quadruples
- *mergelist(*list1, list2*):* returns a new list containing *list1* followed by *list2*
- *backpatch(*list, target*):* inserts the *target* as the target label into each quadruple pointed to by entries in the *list*
- *nextquad():* returns the index of the next quadruple to be generated



Modified Grammar

$$\begin{array}{l} B \rightarrow B \text{ or } MB \\ | B \text{ and } MB \\ | \text{ not } B \\ | (B) \\ | \text{id relop id} \\ | \text{true} \\ | \text{false} \\ M \rightarrow \epsilon \end{array}$$

M is a dummy nonterminal with attribute $M.\text{quad}$, that can hold index of a quadruple

Consider the rule $B \rightarrow B1 \text{ or } MB2$:

Before the reduction of $B2$ starts, reduction $M \rightarrow \epsilon$ has already taken place. Hence, $M.\text{quad}$ points to the index of the first quadruple of $B2$



Translation Rules

$B \rightarrow B1 \text{ or } MB2$

```
{ backpatch(B1.falselist, M.quad)
  B.truelist = mergelist(B1.truelist, B2.truelist)
  B.falselist = B2.falselist
}
```

$B \rightarrow B1 \text{ and } MB2$

```
{ backpatch(B1.truelist, M.quad)
  B.truelist = B2.falselist
  B.falselist = mergelist(B1.falselist, B2.falselist)
}
```

$B \rightarrow \text{not } B1$

```
{ B.truelist = B1.falselist
  B.falselist = B1.truelist
}
```

$B \rightarrow (B1)$

```
{ B.truelist = B1.truelist
  B.falselist = B1.falselist
}
```

$B \rightarrow \text{true}$

```
{ B.truelist = makelist(nextquad())
  emit('goto' ...)
}
```

$B \rightarrow \text{id1 relop id2}$

```
{ B.truelist = nextquad()
  B.falselist = nextquad()
  emit('if' id1.place relop id2.place 'goto' ...)
  emit('goto' ...)
}
```

$B \rightarrow \text{false}$

```
{ B.falselist = makelist(nextquad())
  emit('goto' ...)
}
```

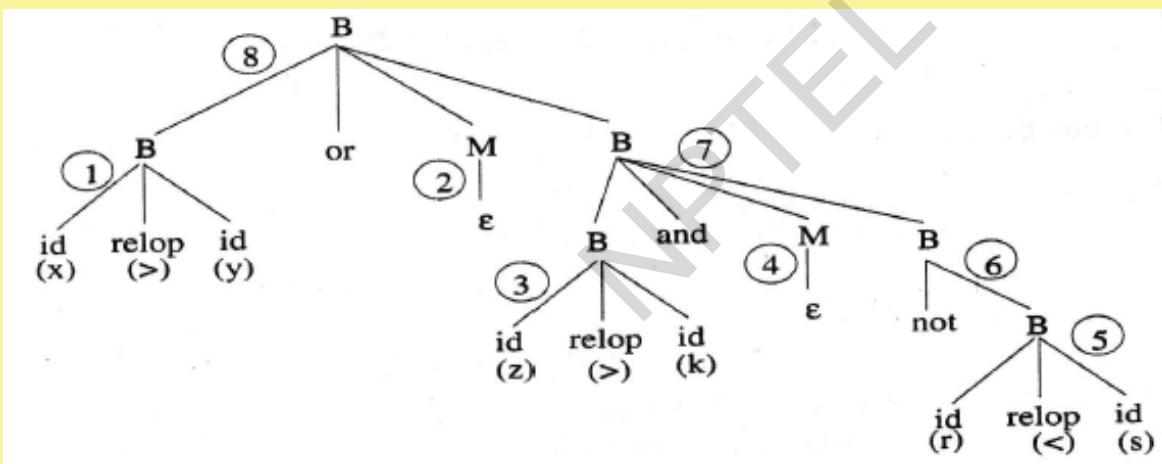
$M \rightarrow \epsilon$

```
{ M.quad = nextquad() }
```



Example

$x > y \text{ or } z > k \text{ and not } r < s$



Translation Example (Contd.)

Reduction	Action	Code generated
1	B.truelist = {1} B.falselist = {2}	1: if x > y goto ... 2: goto ...
2	M.quad = 3	
3	B.truelist = {3}, B.falselist = {4}	3: if z > k goto ... 4: goto ...
4	M.quad = 5	
5	B.truelist = {5} B.falselist = {6}	5: if r > s goto ... 6: goto ...
6	B.truelist = {6}, B.falselist = {5}	
7	Backpatches list {3} with 5	3: if z > k goto 5
8	Backpatches list {2} with 3 B.truelist = {1,6}, B.falselist = {4,5}	2: goto 3

Full Code:

```

1: if x > y goto ...
2: goto 3
3: if z > k goto 5
4: goto ...
5: if r < s goto ...
6: goto ...

```

1, 6 true exit,
4, 5 false exit



Control Flow Statements

- Most programming languages have a common set of statements
 - Assignment: assigns some expression to a variable
 - If-then-else: control flows to either then-part or else-part
 - While-do: control remains within loop until a specified condition becomes false
 - Block of statements: group of statements put within a *begin-end* block marker



Grammar

$S \rightarrow \text{if } B \text{ then } M S$
| $\text{if } B \text{ then } M S N \text{ else } M S$
| $\text{while } M B \text{ do } M S$
| $\text{begin } L \text{ end}$
| $A \quad /* \text{ for assignment } */$

$L \rightarrow L M S$
| S

$M \rightarrow \epsilon$

$N \rightarrow \epsilon$

Attributes:

- $S.\text{nextlist}$: list of quadruples containing jumps to the quadruple following S
- $L.\text{nextlist}$: Same as $S.\text{nextlist}$ for a group of statements

Nonterminal N enables to generate a jump after the *then*-part of *if-then-else* statement.
 $N.\text{nextlist}$ holds the quadruple number for this statement



Translation Rules

$S \rightarrow \text{if } B \text{ then } M_1 S_1$

{ backpatch(B.truelist, M1.quad)

 S.nextlist = mergelist(B.falselist, S1.nextlist)

}

$S \rightarrow \text{if } B \text{ then } M_1 S_1 N \text{ else } M_2 S_2$

{ backpatch(B.truelist, M1.quad)

 backpatch(B.falselist, M2.quad)

 S.nextlist = mergelist(S1.nextlist,
 mergelist(N.nextlist, S2.nextlist))

}

$S \rightarrow \text{while } M_1 B \text{ do } M_2 S_1$

{ backpatch(S1.nextlist, M1.quad)

 backpatch(B.truelist, M2.quad)

 S.nextlist = B.falselist

 emit('goto' M1.quad)

}



Translation Rules

$S \rightarrow \text{if } B \text{ then } M_1 S_1$

{ backpatch(B.truelist, M1.quad)

S.nextlist = mergelist(B.falselist, S1.nextlist)

}

$S \rightarrow \text{if } B \text{ then } M_1 S_1 N \text{ else } M_2 S_2$

{ backpatch(B.truelist, M1.quad)

backpatch(B.falselist, M2.quad)

S.nextlist = mergelist(S1.nextlist,
mergelist(N.nextlist, S2.nextlist))

}

$S \rightarrow \text{while } M_1 B \text{ do } M_2 S_1$

{ backpatch(S1.nextlist, M1.quad)

backpatch(B.truelist, M2.quad)

S.nextlist = B.falselist

emit('goto' M1.quad)

}



Translation Rules (Contd.)

$S \rightarrow \text{begin } L \text{ end}$
{ S.nextlist = L.nextlist }

$S \rightarrow A$
{ S.nextlist = nil }

$L \rightarrow L1 \ M \ S$
{ backpatch(L1.nextlist, M.quad)
L.nextlist = S.nextlist
}

$L \rightarrow S$
{ L.nextlist = S.nextlist }

$M \rightarrow \epsilon$
{ M.quad = nextquad() }

$N \rightarrow \epsilon$
{ N.nextlist = nextquad()
emit('goto' ...)
}



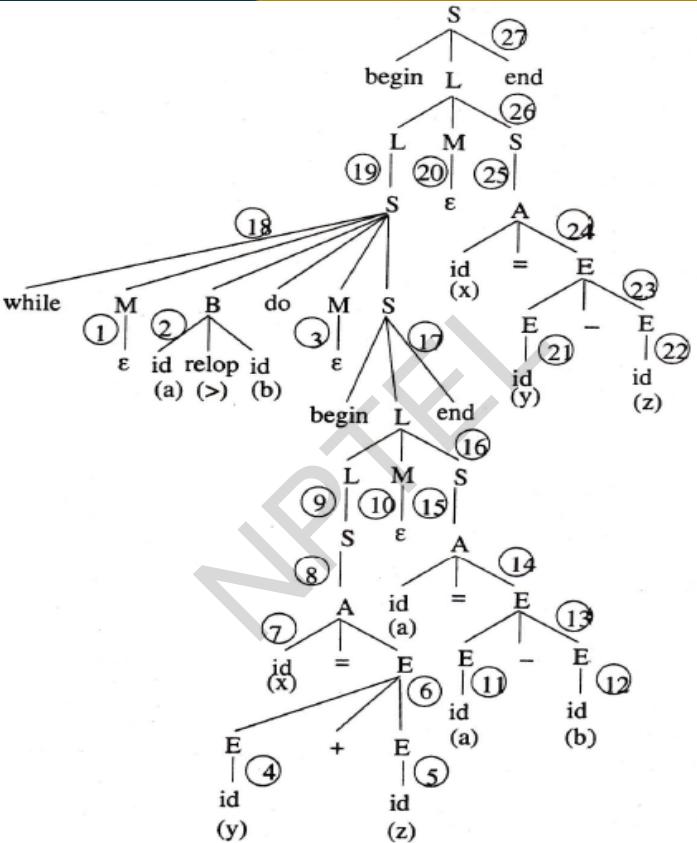
Example

```
begin
    while a > b do
        begin
            x = y + z
            a = a - b
        end
        x = y - z
    end
```

Final Code:

```
1: if a > b goto 3
2: goto 8
3: t1 = y + z
4: x = t1
5: t2 = a - b
6: a = t2
7: goto 1
8: x = t3
```





Red. no.	Action
1	$M.quad = 1$
2	$B.truelist = \{1\}$, $B.falselist = \{2\}$ Code generated: 1: if $a > b$ goto ... 2: goto ...
3	$M.quad = 3$
4	$E.place = y$
5	$E.place = z$
6	$E.place = t_1$ Code generated: 3: $t_1 = y + z$
7	Code generated: 4: $x = t_1$
8	$S.nextlist = \{\}$
9	$L.nextlist = \{\}$
10	$M.quad = 5$
11	$E.place = a$
12	$E.place = b$
13	$E.place = t_2$ Code generated: 5: $t_2 = a - b$
14	Code generated: 6: $a = t_2$
15	$S.nextlist = \{\}$
16	Backpatch($\{\}$, 5) $L.nextlist = \{\}$
17	$S.nextlist = \{\}$
18	backpatch($\{\}$, 1) backpatch($\{1\}$, 3) \Rightarrow Code modified as: 1: if $a > b$ goto 3 $S.nextlist = \{2\}$ Code generated: 7: goto ...
19	$L.nextlist = \{2\}$
20	$M.quad = 8$
21	$E.place = y$
22	$E.place = z$
23	$E.place = t_3$ Code generated: 8: $t_3 = y - z$
24	Code generated: 9: $x = t_3$
25	$S.nextlist = \{\}$
26	Backpatch($\{2\}$, 8) \Rightarrow Code modified as: 2: goto 8 $L.nextlist = \{\}$ $S.nextlist = \{\}$
27	



Case Statements

```
switch(E) {  
    case c1: ...  
    ...  
    case cn: ...  
    default: ...  
}
```

Implementation alternatives:

- Linear search for matching option
- Binary search for matching case
- A jump table
- Linear or binary search may be cheaper if number of cases small, for larger number of cases, jump table may be cheaper
- If case values are not clustered closely together, jump table may be too costly for space



Jump Table Implementation

Let the maximum and the minimum case values be c_{max} and c_{min} respectively

Code to evaluate E into t

```
if  $t < c_{min}$  goto Default_Case  
if  $t > c_{max}$  goto Default_Case  
goto JumpTable[t]  
Default_Case: ...
```

$JumpTable[i]$ is the address of the code to execute, if E evaluates to i



Function Calls

- Can be divided into two subsequences
 - Calling sequence: set of actions executed at the time of calling a function
 - Return sequence: set of actions at the time of returning from the function call
- For both, some actions performed by Caller of the function and the other by the callee



Calling Sequence

Caller

- Evaluate actual parameters
- Place actuals where the callee wants them
- Corresponding three-address instruction:
param t
- Save machine state (current stack and/or frame pointers, return address)
- Corresponding three-address instruction:
call p, n (n=number of actuals)

Callee

- Save registers, if necessary
- Update stack and frame pointers to accommodate m bytes of local storage
- Corresponding three-address instruction:
enter m



Return Sequence

Callee

- Place return value, if any, where the caller wants it
- Adjust stack/frame pointers
- Jump to return address
- Corresponding three-address instruction:
return x or return

Caller

- Save the value returned by the callee
- Corresponding three-address instruction:
retrieve x



Example Function Call

X = f(0, y+1) - 1

```
t1 = y + 1  
param t1  
param 0  
call f, 2  
retrieve t2  
t3 = t2 - 1  
x = t3
```



Storage Allocation for Functions

- Creates problem as the first instruction in a function is:
 enter n /* n = space for locals, temporaries */
- Value of n not known until the whole function has been processed.
- There can be two possible solutions
 - Generating final code in a list
 - Using pair of goto statements

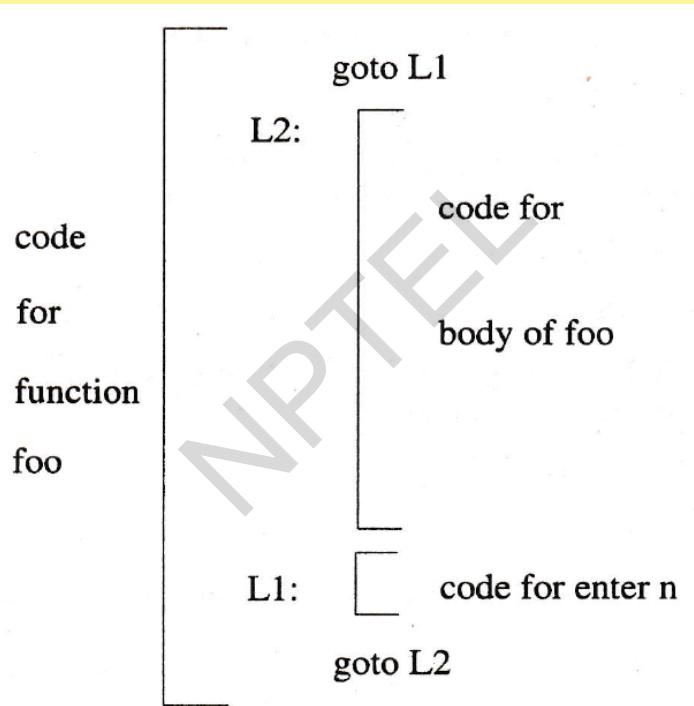


Generating Final Code in List

- Generate final code in a list
- Backpatch the appropriate instructions after processing the function body
- Approach is similar to single-phase code generation for Boolean expressions and control flow statements
- Advantage: Possibility of machine dependent optimizations
- May be slow and may require more memory during code generation



Using Pair of goto Statements



Conclusion

- Intermediate code generation, though not mandatory, helps in retargeting the compiler towards different architectures
- Selecting a good intermediate language itself is a formidable task
- Three-address code is one such representation
- Syntax-directed schemes can be utilized to generate three-address code from the parse tree of the input program
- Translation of almost all major programming language constructs have been carried out





NPTEL ONLINE CERTIFICATION COURSES

Thank
you!