INTRODUCTION

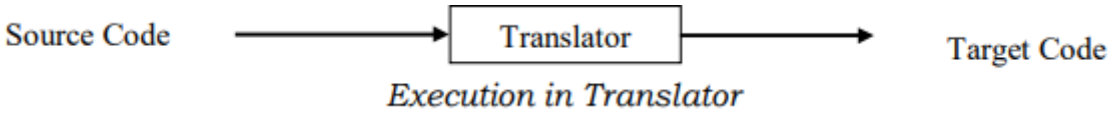
TRANSLATORS

- A translator is a program that takes as input a program written in one language and produces as output a program in another language.
- Beside program translation, the translator performs another very important role, the error-detection. Any violation of d HLL (High Level Language) specification would be detected and reported to the programmers.

Important Role of Translator are:

- Translating the HLL program input into an equivalent ml program.
- Providing diagnostic messages wherever the programmer violates specification of the HLL

A translator or language processor is a program that translates an input program written in a programming language into an equivalent program in another language.



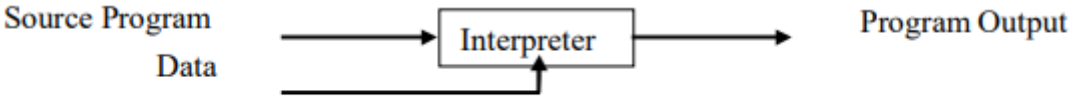
Execution in Translator

Types of Translators:

- Interpreter
- Assembler
- Compiler

Interpreter

An interpreter is a program that appears to execute a source program as if it were machine language. It is one of the translators that translate high level language to low level language.



Execution in Interpreter

During execution, it checks line by line for errors. Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

- Lexical analysis
- Syntax analysis
- Semantic analysis
- Direct Execution

Example: BASIC, Lower Version of Pascal, SNOBOL, LISP & JAVA

Advantages:

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes various may change dynamically.

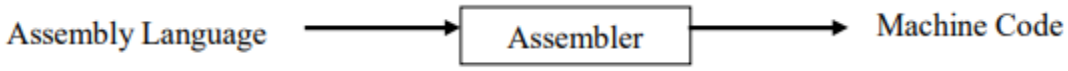
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages:

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

Assembler

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



Example: Microprocessor 8085, 8086.

Advantages:

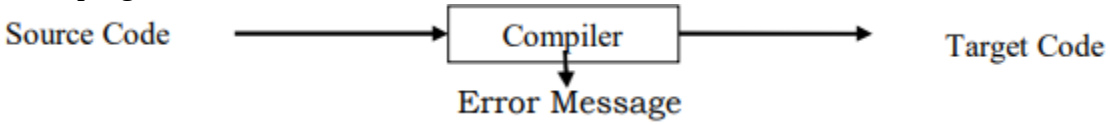
- Debugging and verifying
- Making compilers->Understanding assembly coding techniques is necessary for making compilers, debuggers and other development tools.
- Optimizing code for size.
- Optimizing code for speed.

Disadvantages:

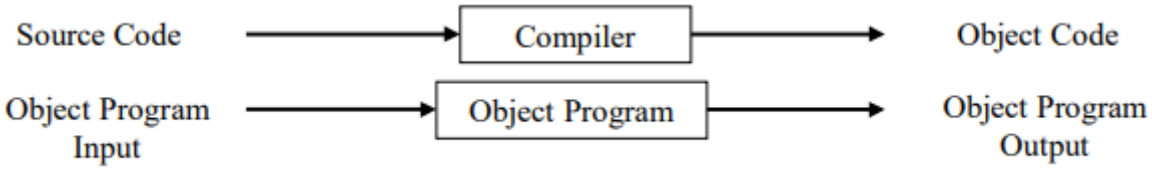
- Development time. Writing code in assembly language takes much longer than writing in a high-level language.
- Reliability and security. It is easy to make errors in assembly code.
- Debugging and verifying. Assembly code is more difficult to debug and verify because there are more possibilities for errors than in high-level code.

Compiler

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important role of a compiler is error showing to the programmer.



Executing a program written in HLL programming language is basically of two parts. The source program must first be compiled translated into an object program. Then the results object program is loaded into a memory executed.

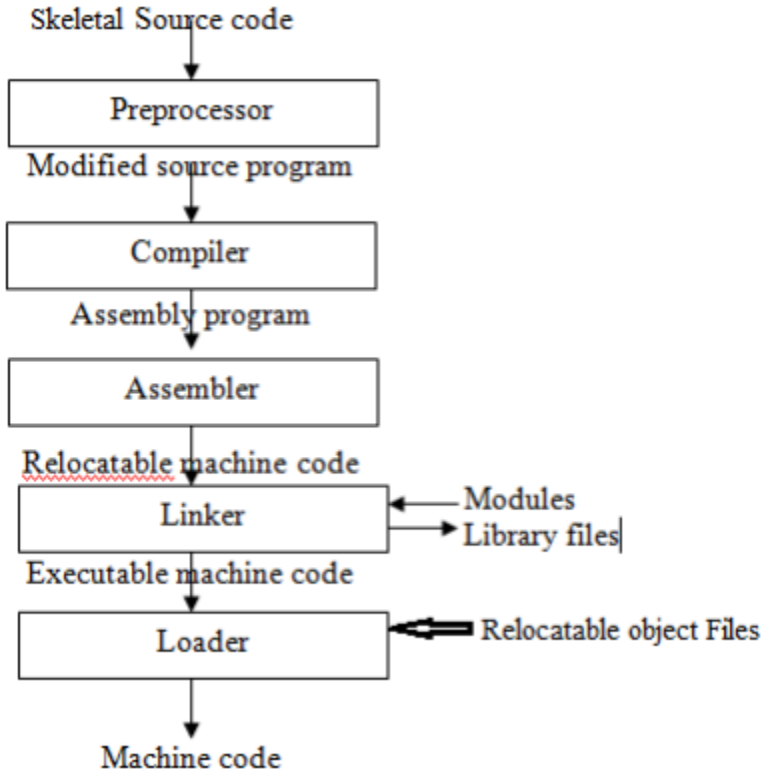


Example: C, C++, COBOL, higher version of Pascal.

Difference between Compiler and Interpreter

Sl.No	Compiler	Interpreter
1	Compiler works on the complete program at once. It takes the entire program as input.	Interpreter program works line-by-line. It takes one statement at a time as input.
2	Compiler generates intermediate code, called the object code or machine code.	Interpreter does not generate intermediate object code or machine code .
3	Compiler executes conditional control statements (like if-else and switch-case) and logical constructs faster than interpreter.	Interpreter executes conditional control statements at a much slower speed.
4	Compiled programs take more memory because the entire object code has to reside in memory.	Interpreter does not generate intermediate object code. As a result, interpreted programs are more memory efficient.
5	Compile once and run anytime. Compiled program does not need to be compiled every time.	5 Interpreted programs are interpreted line-by-line every time they are run.
6	Compiler does not allow a program to run until it is completely error-free.	Interpreter runs the program from first line and stops execution only if it encounters an error.
7	Compiled languages are more efficient but difficult to debug.	Interpreted languages are less efficient but easier to debug. This makes such languages an ideal choice for new students
8	Example: C, C++, COBOL	Example: BASIC, Visual Basic, Python, Ruby, PHP, Perl, MATLAB, Lisp

Language processors (COUSINS OF THE COMPLIER or Language Processing System)



1. Preprocessors:

- It produces input to Compiler. They may perform the following functions.

Macro Processing:

- A preprocessor may allow a user to define macros that are shorthand's for longer constructs.

File inclusion:

- A preprocessor may include header files into the program text. For example, the C preprocessor causes the contents of the file to replace the statement #include when it processes a file containing this statement.

Rational preprocessors:

- These preprocessors augment older language with more modern flow of control and data structuring facilities. If constructs like while-statements does not exist in the programming language, then this preprocessor provides it. Language extensions:
- These preprocessor attempts to add capabilities to the language by what amounts to build in macros. For example, Equal a database query language embedded in C. Statement beginning with ## are taken as preprocessor to be database access statements, unrelated to C and are translated into procedure calls on routines that perform the database access.

2. Complier:

- It converts the source program (HLL) into target program (LLL).

3. Assemblers:

- It converts an assembly language (LLL) into machine code. Some compilers produce assembly for further processing.
- Other compilers perform the job of the assembler, producing relocatable machine code that can be passed directly to the loader/link-editor.
- Assembly code is a mnemonic version of the machine code, in which names are used instead of binary codes for operation and names are also given to memory addresses. A typical sequence of assembly instructions might be

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

4. Loader and Link Editors:

Loader:

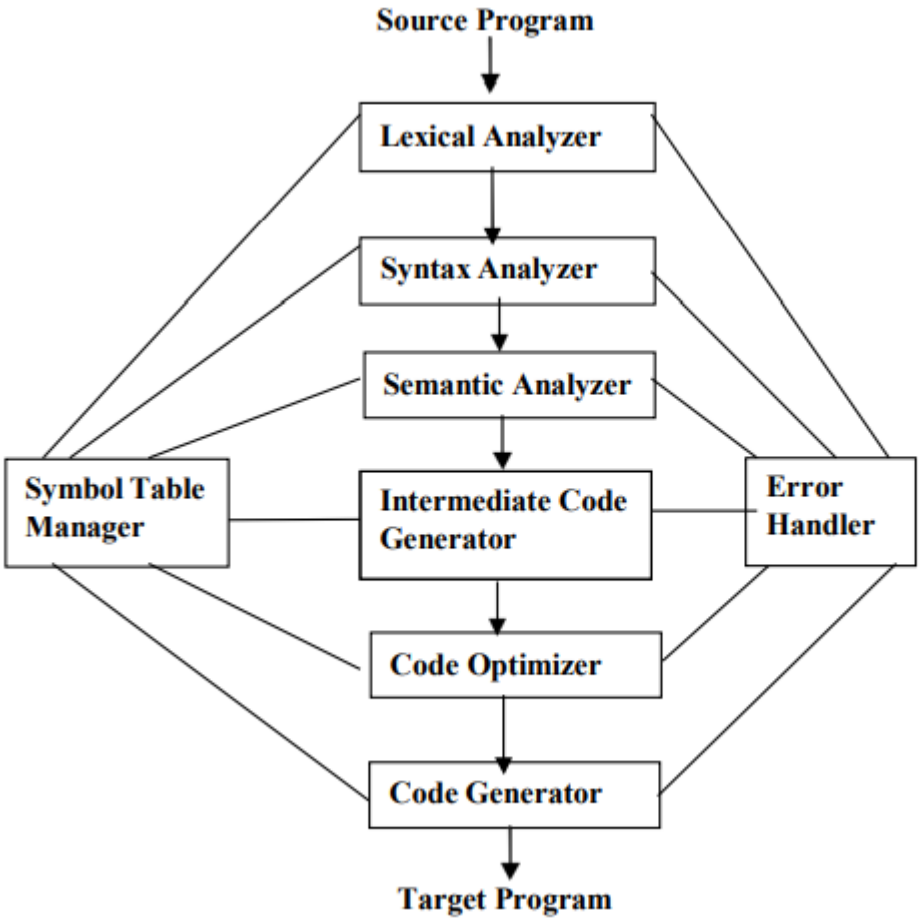
- The process of loading consists of taking relocatable machine code, altering the relocatable addresses and placing the altered instructions and data in memory at the proper locations.
- The Link-editor allows us to make a single program from several files of relocatable machine code. These files may have been the result of several different compilations, and one or more may be library files of routines provided by the system and available to any program that needs them.

Link Editor:

- It allows us to make a single program from several files of relocatable machine code.

THE PHASES OF COMPILER

1) Lexical analysis - it contains a sequence of characters called tokens. Input is source program & the output is tokens.
2) Syntax analysis - input is token and the output is parse tree
3) Semantic analysis - input is parse tree and the output is expanded version of parse tree
4) Intermediate Code generation - Here all the errors are checked & it produce an intermediate code.
5) Code Optimization - the intermediate code is optimized here to get the target program.
6) Code Generation - this is the final step & here the target program code is generated.



Lexical Analysis:

In a compiler, Linear analysis is called lexical analysis or scanning. For example, in lexical analysis the characters in the assignment statement

position: = initial + rate * 60

would be grouped in to the following tokens

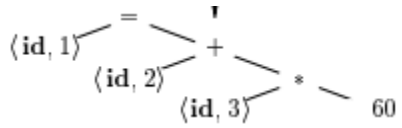
1. The identifier position
2. The assignment symbol: =
3. The identifier initial
4. The plus sign
5. The identifier rate
6. The multiplication sign

7. The number 60

The blanks are usually eliminated during lexical analysis

Syntax Analysis:

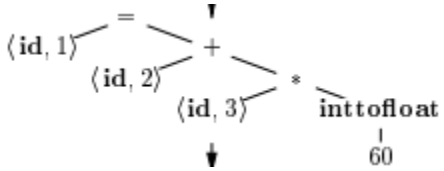
The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.



A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

Semantic Analysis:

The semantic analysis checks the source program for semantic errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.



Intermediate Code Generator:

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation can have a variety of forms. In three-address code, the source program might look like this,

temp1: = inttoreal (60)

temp2: = id3 * temp1

temp3: = id2 + temp2

id1: = temp3

Code Optimization:

The code optimization phase attempts to improve the intermediate code, so that faster running machine codes will result. Some optimizations are trivial. There is a great variation in the amount of code optimization different compilers perform. In those that do the most, called 'optimizing compilers', a significant fraction of the time of the compiler is spent on this phase.

temp1 := id3 * 60.0

id1 := id2 + temp1

Code Generation:

The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program.

MOVF id3, R2

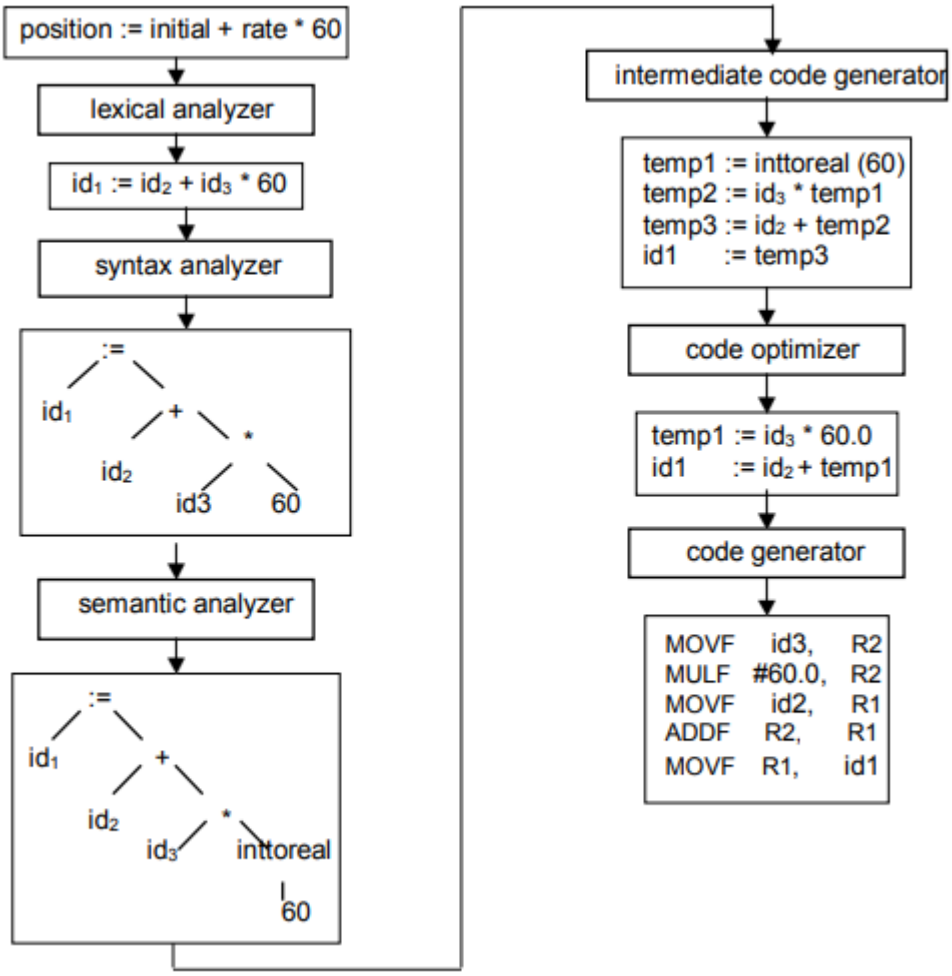
MULF #60.0, R2

MOVF id2, R1

ADDF R2, R1

MOVF R1, id1

The first and second operands of each instruction specify a source and destination, respectively. The F in each instruction tells us that instruction deal with floating point numbers.



Symbol-Table Management:

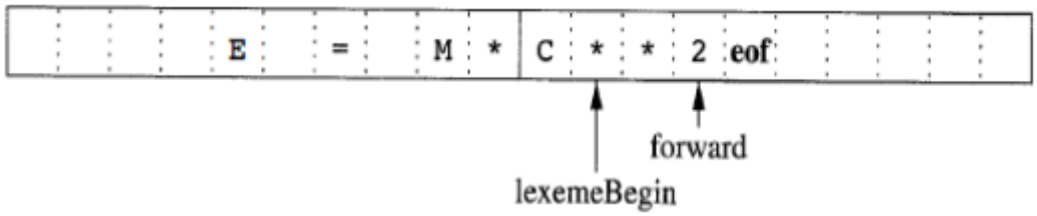
The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

INPUT BUFFERING

- We often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.
- As characters are read from left to right, each character is stored in the buffer to form a meaningful token. We introduce a two-buffer scheme that handles large look a head's safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

Buffer Pairs:

- A buffer is divided into two N-character halves, as shown below.
- Each buffer is of the same size N, and N is usually the number of characters on one disk block. E.g., 1024 or 4096 bytes.
- Using one system read command we can read N characters into a buffer.
- If fewer than N characters remain in the input file, then a special character, represented by eof, marks the end of the source file.

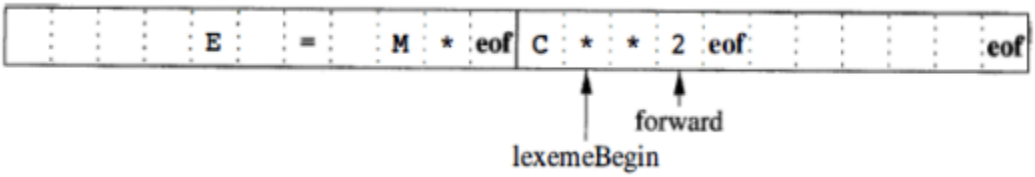


Two pointers to the input are maintained:

- Pointer lexeme_beginning, marks the beginning of the current lexeme, whose extent we are attempting to determine.
- Pointer forward scans ahead until a pattern match is found.

Sentinels:

- For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character eof.
- The sentinel arrangement is as shown below:



SPECIFICATION OF TOKEN:

- Regular expressions are notation for specifying patterns.
- Each pattern matches a set of strings.
- Regular expressions will serve as names for sets of strings.
- Strings and Languages String means a finite sequence of symbols.

For example

computer (c, o, m, p, u, t, e, r)
CS6660 (C, S, 6, 6, 6, 0)
101001 (1, 0)

- Symbols are given through alphabet. An alphabet is a finite set of symbols
- The term alphabet or character class denotes any finite set of symbols. e.g., set {0,1} is the binary alphabet.
- The term sentence and word are often used as synonyms for the term string.
- The length of a string s is written as | s | - is the number of occurrences of symbols
- The empty string denoted by ε – length of empty string is zero.
- The term language denotes any set of strings over some fixed alphabet.

Operations on Languages:

There are several operations that can be applied to languages:

Definitions of operations on languages L and M:

OPERATION	DEFINITION
Union of L and M. written LυM	L υ M = { s | s is in L or s is in M }
Concatenation of L and M. written LM	LM = { st | s is in L and t is in M }
Kleene closure of L. written L*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L* denotes "zero or more concatenation of" L.
Positive closure of L. written L+	$L^+ =$ L^+ denotes "one or more Concatenation of" L.

Regular Expressions:

- It allows defining the sets to form tokens.
- Defines a Pascal identifier –identifier is formed by a letter followed by zero or more letters or digits. e.g., letter (letter | digit) *
- A regular expression is formed using a set of defining rules.
- Each regular expression r denotes a language L(r).

Order of evaluate Regular expression:

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

- The unary operator * has highest precedence and is left associative.
- Concatenation has second highest precedence and is left associative.
- | has lowest precedence and is left associative.

RECOGNITION OF TOKENS:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

Grammar for branching statements:

Stmt →if expr then stmt

| If expr then else stmt

| ϵ

Expr →term relop term

| term

Term →id

|number

The terminal of grammar, which are if, then, else, relop,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

FINITE AUTOMATA

- Finite automata are a mechanism to recognize a set of valid inputs before carrying out an action.
- Finite automata are a mathematical model of a system with inputs, outputs, finite number of states and a transition from state to state on input symbol Σ.
- A recognizer for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a finite automaton. A finite automaton can be: deterministic (DFA) or non-deterministic (NFA). Both deterministic and nondeterministic finite automaton recognize regular sets.

Non Deterministic Finite Automata:

NFA (Non-deterministic Finite Automaton) is a 5-tuple (S, Σ, δ, S0, F):

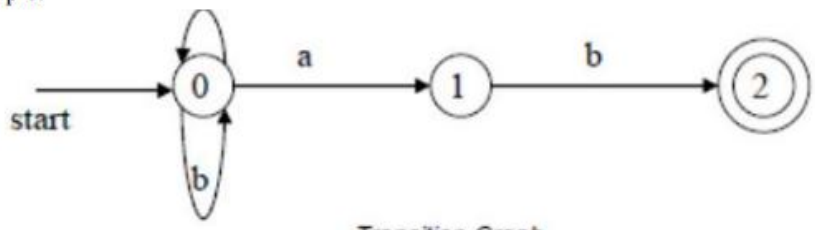
- S: a set of states
- Σ: the symbols of the input alphabet
  - δ is a set of transition function
- S0: s0 ∈S, the start state
- F: F ⊆ S, a set of final or accepting states.

Transition table

A transition table is a good way to implement a FSA

- One row for each state, S
- One column for each symbol, A
- Entry in cell (S,A) gives the state or set of states can be reached from state S on input A.

Example:



Transition Graph

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

Transition Function:

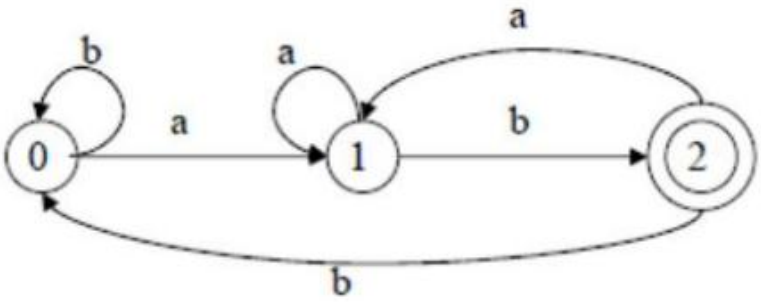
	a	b
0	{0,1}	{0}
1	{}	{2}
2	{}	{}

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

Deterministic Finite Automaton (DFA):

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

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



Transition Graph

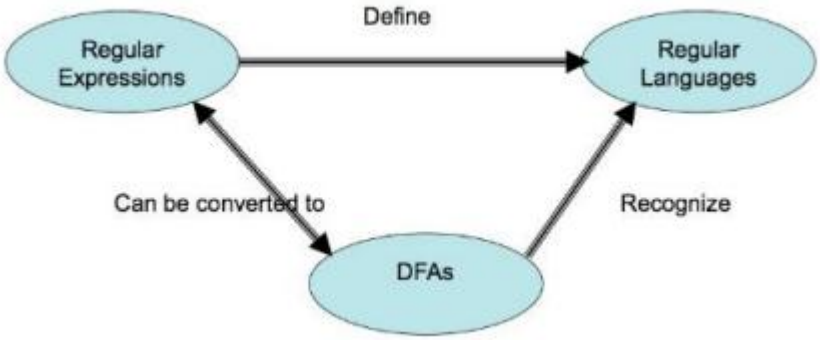
0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

Transition Function:

	a	b
0	1	0
1	1	2
2	1	0

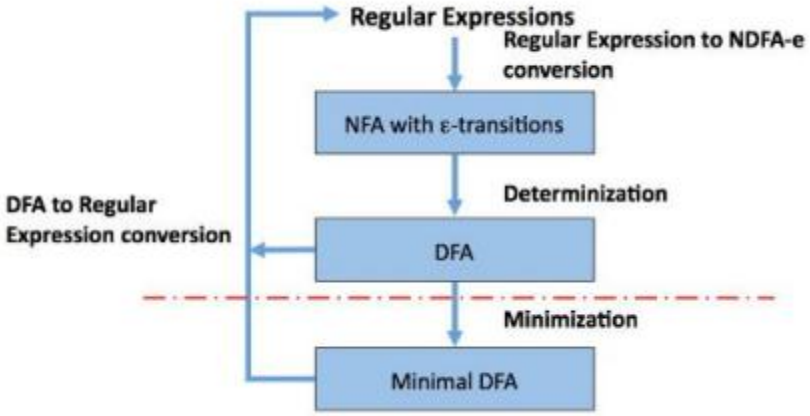
Construction of an NFA and DFA from a Regular Expression:

- To convert a regular expression to a NFA using McNaughton-Yamada-Thompson algorithm
- syntax-directed: It works recursively up the parse tree of the regular expression
- For each sub expression a NFA with a single accepting state is built



Conversion of a NFA to a DFA:

- Subset construction: Each state of DFA corresponds to a set of NFA states.
- DFA states may be exponential in number of NFA states.

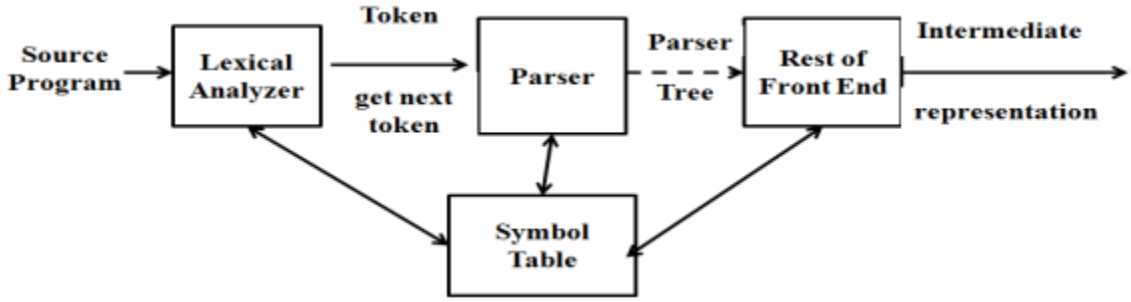


Operations on NFA states:

Operation	Description
ε-closure(s)	set of NFA states reachable from NFA state *s* on ε- transition alone
ε-closure(T)	set of NFA states reachable from some NFA state *s* in set T on ε-transitions alone
move(T,a)	set of NFA states to which there is a transition on input symbol a from some state s in T

ROLE OF PARSER

- The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language.
- It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.



Role of the Parser:

- Parser builds the parse tree.
- Parser verifies the structure generated by the tokens based on the grammar (performs context free syntax analysis).
- Parser helps to construct intermediate code.
- Parser produces appropriate error messages.
- Parser performs error recovery.

Issues:

- Parser cannot detect errors such as:
- Variable re-declaration
- Variable initialization before use
- Data type mismatch for an operation

GRAMMARS

- Constructs that begin with keywords like while or int, are relatively easy to parse, because the keyword guides the choice of the grammar production that must be applied to match the input. Consider the grammar for expressions, which present more of challenge, because of the associativity and precedence of operators.
- In the following expression grammar E represents expressions consisting of terms separated by + signs, T represents terms consisting of factors separated by * signs, and F represents factors that can be either parenthesized expressions or identifiers.

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

Error Handling:

Programs can contain errors at many different levels.

For example:

- Lexical, such as misspelling an identifier, keyword or operators

- Syntactic, such as misplaced semicolons or extra or missing braces, a case statement without an enclosing switch statement.
- Semantic, such as type mismatches between operators and operands.
- Logical, such as an assignment operator = instead of the comparison operator ==, infinitely recursive call.

Functions of error handler:

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

Error-Recovery Strategies:

Once an error is detected, the parser should recover from the error. The following recovery strategies are

- **Panic-mode**: On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or}, whose role in the source program is clear and unambiguous
- **Phrase-level**: When an error is identified, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon. Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string. Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection
- **Error-productions**: The common errors that might be encountered are anticipated and augment the grammar for the language at hand with productions that generate the erroneous constructs.
- **Global-correction**: A compiler need to make few changes as possible in processing an incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible. These methods are in general too costly to implement in terms of time and space.

CONTEXT-FREE GRAMMARS

Grammars are used to systematically describe the syntax of programming language constructs like expressions and statements. Using a syntactic variable stmt to denote statements and variable expr to denote expressions, the production

stmt → **if** (expr) stmt **else** stmt

It specifies the structure of this form of conditional statement.

The Formal Definition of a Context-Free Grammar

A context-free grammar G is defined by the 4-tuple: G= (V, T, P S) where

1. V is a finite set of non-terminals (variable).
2. T is a finite set of terminals.
3. S is the start symbol (variable S∈V)
4. The productions of a grammar specify the manner in which the terminals and nonterminals can be combined to form strings.

Notational Conventions

These symbols are terminals:

- Lowercase letters early in the alphabet, such as a, b, c.
- Operator symbols such as +, *,- , / and so on.
- Punctuation symbols such as parentheses, comma, and so on.
- The digits 0 , 1 , . . . ,9.
- Boldface strings such as id or if, each of which represents a single terminal symbol

These symbols are non-terminals:

- Uppercase letters early in the alphabet, such as A, B, C.
- The letter S, which, when it appears, is usually the start symbol.
- Lowercase, italic names such as expr or stmt.

Example:

E → E + T | E - T | T
T → T * F | T / F | F
F → (E) | **id**

Start symbol: E

Terminal: +, - , * , / , (,) , id

Non Terminal: E , T , F

Derivations:

`Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example: Consider the following grammar for arithmetic expressions:

E → E+E | E*E | (E) | - E | id

To generate a valid string - (id+id) from the grammar the steps are

1. E → - E
2. E → - (E)
3. E → - (E+E)
4. E → - (id+E)
5. E → - (id+id)

In the above derivation,

- E is the start symbol.
- -(id+id) is the required sentence (only terminals).
- Strings such as E, -E, -(E), . . . are called **sentinel** forms.

Types of derivations:

The two types of derivation are:

- Left most derivation: In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- Right most derivation: In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement. Rightmost derivations are sometimes called canonical derivations.

Example: Given grammar G : E → E+E | E*E | (E) | - E | id
Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

Parse Trees and Derivations:

- A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.
- Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.

Ambiguity:

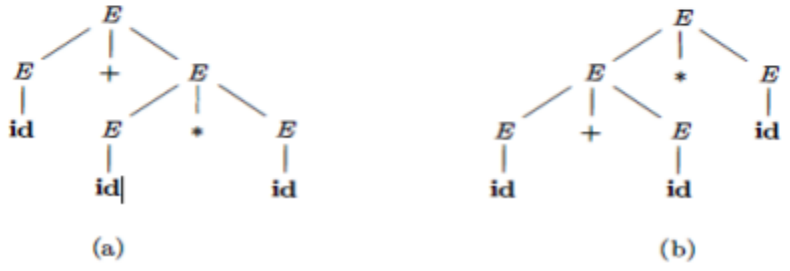
- A grammar produce more than one parse tree for some sentence is said to be ambiguous. A grammar G is said to be ambiguous if it has more than one parse tree either in LMD or in RMD for at least one string.

Example: Given grammar G : E → E+E | E*E | (E) | - E | id

The sentence id+id*id has the following two distinct leftmost derivations:

Leftmost Derivations -1	***Leftmost Derivations -2***
E → E+ E	E → E* E
E → id + E	E → E + E * E
E → id + E * E	E → id + E * E
E → id + id * E	E → id + id * E
E → id + id * id	E → id + id * id

The two corresponding parse trees are :



WRITING A GRAMMAR

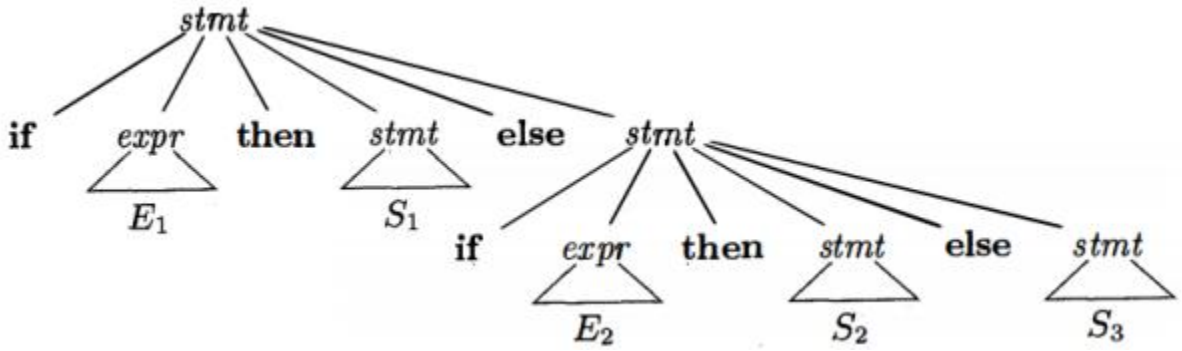
- Grammars are capable of describing most, but not all, of the syntax of programming languages. For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar.
- Therefore, the sequences of tokens accepted by a parser form a superset of the programming language; subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

Eliminating Ambiguity

Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar. Consider the following "dangling else" grammar.

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

Here "other" stands for any other statement. According to this grammar, the compound conditional statement if E1 then S1 else if E2 then S2 else S3

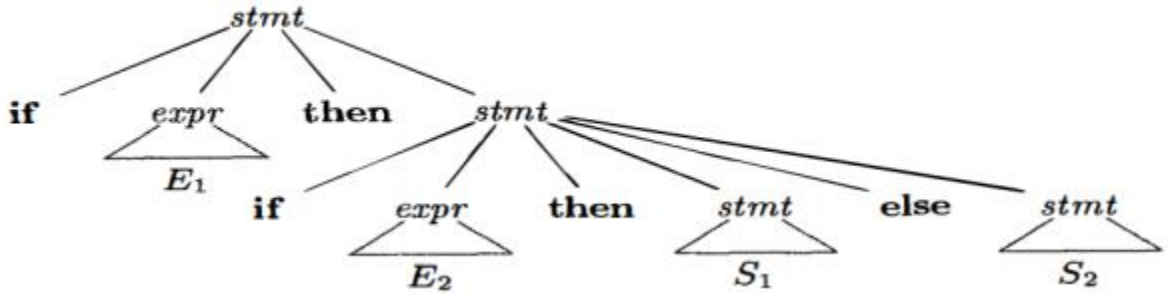


But the grammar is ambiguous since the string

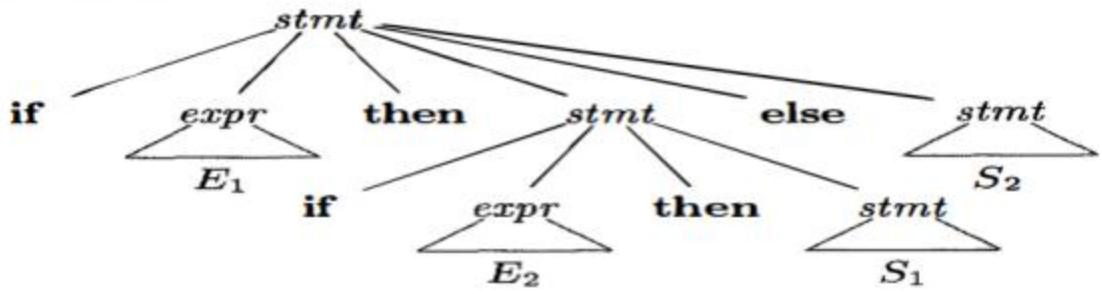
if E1 then if E2 then S1 else S2

Has the following two parse trees for leftmost derivation:

Parse Tree-1



Parse Tree-2



Elimination of Left Recursion:

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

  A=> Aα
- Top down parsing methods can't handle left-recursive grammars
- A simple rule for immediate left recursion elimination for: A => A α|β
- We may replace it with

  A -> β A'

  A' -> α A' | ε

Example to eliminate Immediate left recursion: Consider the following grammar for arithmetic expressions:

E → E+T | T

T → T*F | F

F → (E) | id

Answer: Eliminate left recursive productions E and T by applying the left recursion

First eliminate the left recursion for E as

E → TE'

E' → +TE' | ε

Then eliminate for T as

T → FT'

T'→ *FT'| ε

Thus the obtained grammar after eliminating left recursion is

E → TE'

E' → +TE'| ε

T → FT'

T' → *FT'| ε

F → (E) | id

Left factoring

- Left factoring is a process of factoring out the common prefixes of two or more production alternates for the same nonterminal.
- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing.

Consider following grammar:

stmt -> 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. So, can perform left factoring, where the general form is:

If we have **A → αβ1 | αβ2** then we replace it with

A → αA'

A' → β1 | β2

Example: Eliminate left factors from the given grammar. S → T + S | T

After left factoring, the grammar becomes,

S →T S'

S'→ + S | ε

Example: Left factor the following grammar.

S → aBcDeF | aBcDgg | F , B→ x, D→ y, F→ z

After left factoring, the grammar becomes,

S → aBcDS' | F

S' → eF | gg

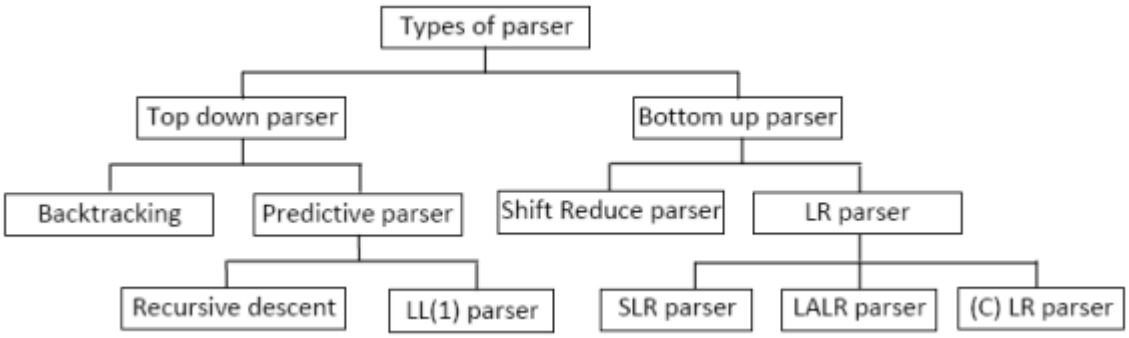
B→ x

D→ y

F→ z

TOP DOWN PARSING

- Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first).
- Top down parsing can be viewed as finding a leftmost derivation for an input string.
- Parsers are generally distinguished by whether they work top-down (start with the grammar's start symbol and construct the parse tree from the top) or bottom-up (start with the terminal symbols that form the leaves of the parse tree and build the tree from the bottom).
- Top down parsers include recursive-descent and LL parsers, while the most common forms of bottom up parsers are LR parsers



RECURSIVE DESCENT PARSER

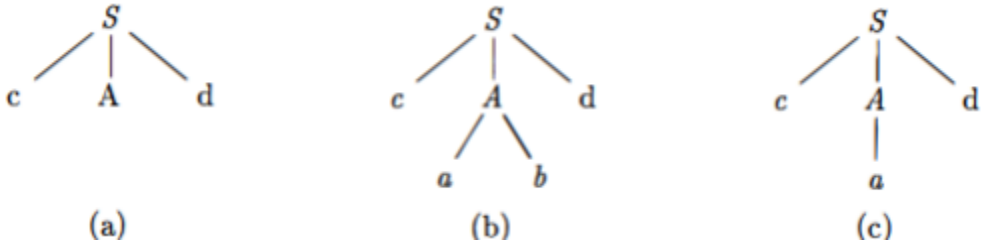
- These parsers use a procedure for each nonterminal. The procedure looks at its input and decides which production to apply for its nonterminal.
- Terminals in the body of the production are matched to the input at the appropriate time, while nonterminals in the body result in calls to their procedure.
- General recursive-descent may require backtracking; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs.

Example : Consider the grammar

S→cAd

A→ ab | a

To construct a parse tree top-down for the input string w = cad



(a) (b) (c)

Step 1:

- Initially create a tree with single node labeled S. An input pointer points to ‘c’, the first symbol of w. Expand the tree with the production of S.

An input pointer points to ‘c’, the first symbol of w. Expand the tree with the production of S

Step 2:

- The leftmost leaf ‘c’ matches the first symbol of w, so advance the input pointer to the second symbol of w ‘a’ and consider the next leaf ‘A’.

Step 3:

- The second symbol ‘a’ of w also matches with second leaf of tree. So advance the input pointer to third symbol of w ‘d’. But the third leaf of tree is b which does not match with the input symbol d.

Hence discard the chosen production and reset the pointer to second position. This is called backtracking.

FIRST( ) & FOLLOW( ):

- First(α) is defined as set of terminals that begins strings derived from α.
- If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ε is also in First(α)
- In predictive parsing when we have A-> α|β, 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.

Algorithm to compute FIRST(X)

To compute **FIRST(X)** for all grammar symbols X, apply the following rules until no more terminals or ε can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = {X}.
2. If X is a nonterminal and X → $Y_1Y_2...Y_k$ is a production for some k≥1, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and **ε** is in all of FIRST(Y_1),..., FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. (If **ε** is in FIRST($Y_j$) for all j = 1,2, . . . , k, then add **ε** to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive **ε**, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add F1RST($Y_2$), and So on.
3. If X→ **ε** is a production, then add **ε** to FIRST(X).

Algorithm to compute FOLLOW(A)

To compute **FOLLOW(A)** for all non-terminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right endmarker.
2. If there is a production A→αBβ, then everything in FIRST(β) except **ε** is in FOLLOW(B).
3. If there is a production A→αB, or a production A→αBβ, where FIRST(β) contains **ε**, then everything in FOLLOW (A) is in FOLLOW (B).

Example : To compute FIRST and FOLLOW

Consider again the non-left-recursive grammar

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

- FIRST(F) = FIRST(T) = FIRST(E) = {(, id}. To see why, note that the two productions for F have bodies that start with these two terminal symbols, id and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ε, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).
- FIRST(E') = {+, ε }. The reason is that one of the two productions for E' has a body that begins with terminal +, and the other's body is ε. Whenever a nonterminal derives ε, we place ε in FIRST for that nonterminal.
- FIRST(T') = {* , t}. The reasoning is analogous to that for FIRST(E').
- FOLLOW(E) = FOLLOW(E') = {), $}. Since E is the start symbol, FOLLOW(E) must contain $. The production body (E) explains why the right parenthesis is in FOLLOW(E) . For E' , note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E') must be the same as FOLLOW(E).
- FOLLOW(T) = FOLLOW(T') = {+,), $}. Notice that T appears in bodies only followed by E' . Thus, everything except ε that is in FIRST(E') must be in FOLLOW(T) ; that explains the symbol +.

PREDICTIVE PARSER – LL(1) PARSER

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.

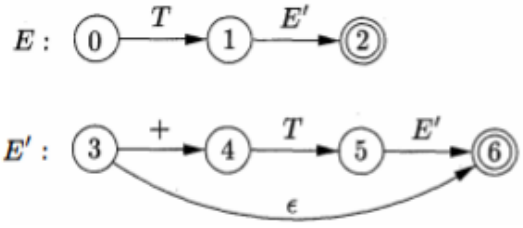
- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.
- This parser looks up the production in parsing table.
- The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

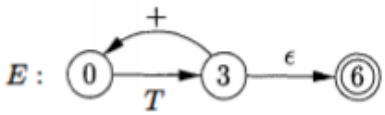
Transition Diagrams for Predictive Parsers:

- Transition diagrams are useful for visualizing predictive parsers.
- For example, the transition diagrams for nonterminals E and E' of expression grammar

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

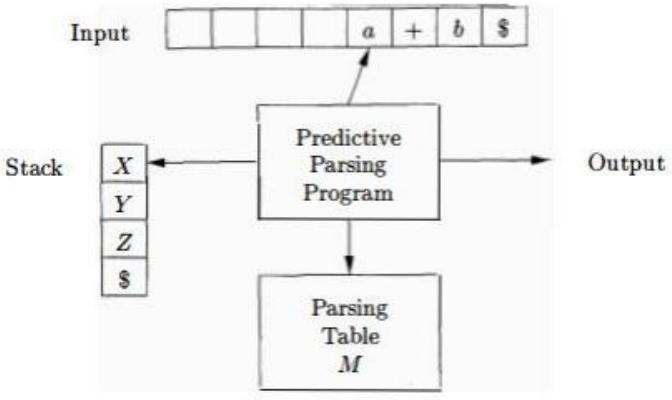
- To construct the transition diagram from a grammar, first eliminate left recursion and then left factor the grammar.
- Transition diagrams for predictive parsers differ from those for lexical analyzers.
- Parsers have one diagram for each nonterminal.
- The labels of edges can be tokens or nonterminals.
- A transition on a token (terminal) means that we take that transition if that token is the next input symbol.





Predictive Parsing Program:

The predictive parser has an input, a stack, a parsing Table and an output.



Input: Contains the string to be parsed, followed by right end marker $.

Stack: Contains a sequence of grammar symbols, preceded by $, the bottom of stack marker. Initially the stack contains the start symbol of the grammar preceded by $.

Parsing Table: It is a two dimensional array M[A,a], where A is a non-terminal and a is a terminal or $.

Output: Gives the output whether the string is valid or not.

INTRODUCTION TO LALR PARSER

- The last parser method LALR (LookAhead-LR) technique is often used in practice because the tables obtained by it are considerably smaller than the canonical LR tables.
- The SLR and LALR tables for a grammar always have the same number of states whereas CLR would typically have several number of states.
- For example, for a language like Pascal SLR and LALR would have hundreds of states but CLR would have several thousands of states.

Algorithm: LALR table construction.

INPUT: An augmented grammar G'.

OUTPUT: The LALR parsing-table functions ACTION and GOT0 for G'.

METHOD:

1. Construct C = (I0 I1, . . , In), the collection of sets of LR(1) items.

2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.

3. Let C' = {J0J1, . . . ,Jm) be the resulting sets of LR(1) items. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

4. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is, J = I1 U I2 U . . .U Ik , then the cores of GOTO(I1, X) , GOTO(I2, X) , . . . , GOTO(Ik, X) are the same, since I1,I2, . . ,Ik all have the same core. Let K be the union of all sets of items having the same core as GOTO(I1, X). Then GOTO(J,X) = K.

Let us consider grammar

S->CC

C->cC

C->d whose sets of LR(1) items are

I_0 : S'->.S , $ S->.CC, $ C->.cC , c/d C->.d , c/d Goto(I_0,S) I_1 : S'->S., $ Goto(I_0, C) I_2 : S->C.C,$ C-> .cC,$ C->.d ,$	Goto(I_0, c) I_3: C->c.C, c/d C->.cC, c/d C-> .d , c/d Goto(I_0,d) I_4: C->d., c/d Goto(I_2, C) I_5: S->CC., $ Goto(I_2,c) I_6: C->c.C, $ C-> .cC, $ C->.d, $	Goto(I_2,d) I_7: C ->d., $ Goto(I_3, C) I_8: C->cC. , c/d Goto(I_3,c) = I_3 Goto(I_3, d) = I_4 Goto (I_6, C) I_9: C->cC., $ Goto(I_6, c) = I_6 Goto(I_6, d) = I_7

In the above LR(1) items, there are three pairs of sets of items that can be merged. I_3 and I_6 are replaced by their union:

I_{36}: C->c.C, c/d/$
C->.cC, c/d /$
C-> .d , c/d/$

I_4 and I_7 are replaced by their union:

I_{47}: C->d., c/d/$

and I_8 and I_9 are replaced by their union

I_8: C->cC. , c/d/$

The LALR action and goto functions for the condensed sets of items are

STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Error recovery in LR parsing:

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. An LR parser will announce an error as soon as there is no valid continuation for the portion of the input thus far scanned.
- A canonical LR parser will not make even a single reduction before announcing an error. SLR and LALR parsers may make several reductions before announcing an error, but they will never shift an erroneous input symbol onto the stack.

Panic-mode error recovery:

- We scan down the stack until a state s with a goto on a particular nonterminal A is found. Zero or more input symbols are then discarded until a symbol a is found that can legitimately follow A.
- The parser then stacks the state GOTO (s, A) and resumes normal parsing.

Phrase-level recovery:

- It is implemented by examining each error entry in the LR parsing table and deciding on the basis of language usage the most likely programmer error that would give rise to that error.
- An appropriate recovery procedure can then be constructed; presumably the top of the stack and/or first input symbols would be modified in a way deemed appropriate for each error entry.

ERROR HANDLING AND RECOVERY IN SYNTAX ANALYZER

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

Panic mode recovery:

- On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or end.
- It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

Phrase level recovery:

- On discovering an error, the parser performs local correction on the remaining input that allows it to continue.
- Example: Insert a missing semicolon or delete an extraneous semicolon etc.

Error productions:

- The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

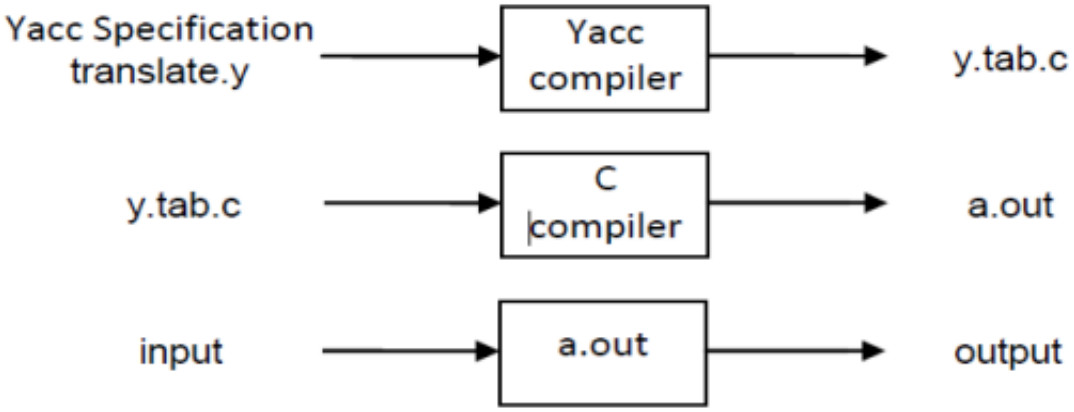
Global correction:

- Given an incorrect input string x and grammar G, certain algorithms can be used to find a parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

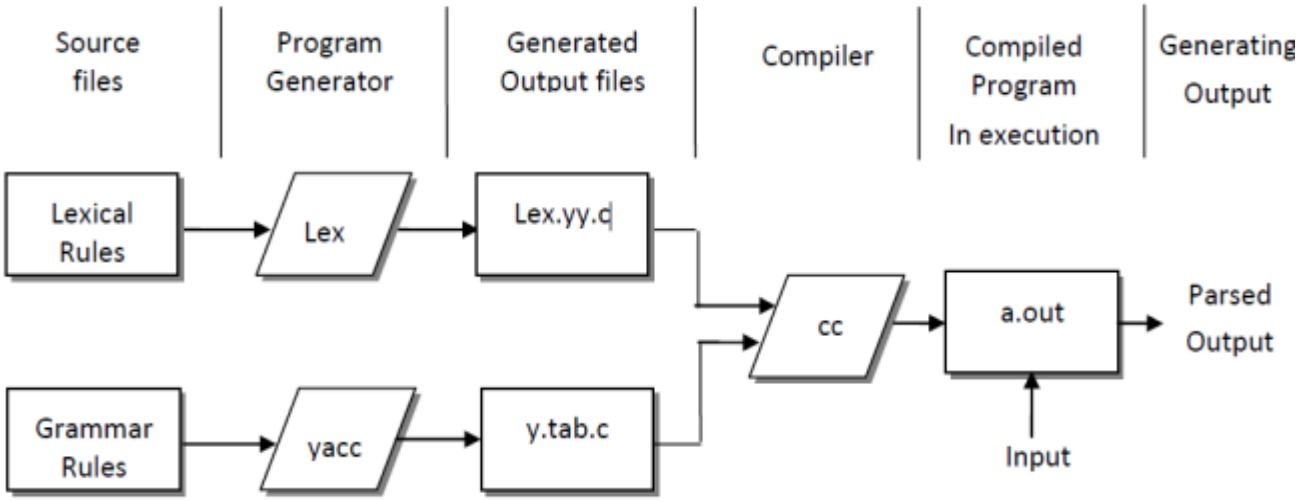
YACC

- YACC is a Yet Another Compiler Compiler.
- Yacc is a computer program for the Unix operating system.
- It is a LALR parser generator, generating a parser, the part of a compiler that tries to make syntactic sense of the source code, specifically a LALR parser, based on an analytic grammar written in a notation similar to BNF.
- Yacc itself used to be available as the default parser generator on most Unix systems.
- The input to Yacc is a grammar with snippets of C code (called "actions") attached to its rules.
- Its output is a shift-reduce parser in C that executes the C snippets associated with each rule as soon as the rule is recognized. Typical actions involve the construction of parse trees.

declarations
%%
translation rules
%%
supporting C routine



Design of a syntax analyzer for a sample language:



YACC (Yet Another Compiler Compiler).

- Automatically generate a parser for a context free grammar (LALR parser)
  Allows syntax direct translation by writing grammar productions and semantic action
  LALR(1) is more powerful than LL(1).
- Work with lex. YACC calls yylex to get the next token.
  YACC and lex must agree on the values for each token.
- Like lex, YACC pre-dated c++, need workaround for some constructs when using c++ (will give an example).
- yyparse() returns 0 if the program is grammatically correct, non-zero otherwise.
- YACC automatically builds a parser for the grammar (LALR parser).

Program to recognize a valid variable (identifier) which starts with a letter followed by any number of letters or digits.

```
LEX
%{
        #include"y.tab.h"
        extern   yylval;
%}
%%
[0-9]+    {yylval=atoi(yytext);  return  DIGIT;}
[a-zA-Z]+         {return LETTER;}
[\t]     ;
\n      return 0;
.       {return yytext[0];}
%%
YACC
%{
        #include<stdio.h>
%}
%token           LETTER DIGIT
%%
variable:        LETTER|LETTER rest
;
rest:  LETTER rest
        |DIGIT rest
        |LETTER
        |DIGIT
        ;
%%
main()
{
        yyparse();
        printf("The string is a valid variable\n");
}
int yyerror(char *s)
{
        printf("this is not a valid variable\n");
        exit(0);
}
```

OUTPUT

```
$lex p4b.l
$yacc –d p4b.y
$cc lex.yy.c y.tab.c –ll
$./a.out
input34
The string is a valid variable
$./a.out
89file
This is not a valid variable
```

SYNTAX DIRECTED DEFINITION

- A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions.
- For example, an infix-to-postfix translator might have a production and rule.

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

- This production has two non-terminals, E and T; the subscript in El distinguishes the occurrence of E in the production body from the occurrence of E as the head; Both E and T have a string-valued attribute code.
- The semantic rule specifies that the string E. code is formed by concatenating E1 .code, T. code, and the character ' +'.
- A syntax-directed translation scheme embeds program fragments called semantic actions within production bodies, as in

$$E \rightarrow E_1 + T \quad \{ \text{ print } '+' \ \}$$

- The most general approach to syntax-directed translation is to construct a parse tree or a syntax tree, and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree.

SDD:

- A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.
- If X is a symbol and a is one of its attributes, then we write X.a to denote the value of a particular parse-tree node labeled X.

Inherited and Synthesized Attributes

1. Synthesized attribute:

- A synthesized attribute for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N.
- A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.

2. Inherited Attribute:

- An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N.
- An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

Example:

The SDD in Figure below is based on arithmetic expressions with operators + and *. It evaluates expressions terminated by an end marker n. In the SDD, each of the nonterminals has a single synthesized attribute, called val. We also suppose that the terminal digit has a synthesized attribute lexval, which is an integer value returned by the lexical analyzer.

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

- The rule for production 1, L -> E n, sets L.val to E.val, which we shall see is the numerical value of the entire expression. Production 2, E -> El + T, also has one rule, which computes the val attribute for the head E as the sum of the values at El and T.
- At any parse tree node N labeled E, the value of val for E is the sum of the values of val at the children of node N labeled E and T.
- An SDD that involves only synthesized attributes is called S-attributed;
- In an S-attributed SDD, each rule computes an attribute for the non-terminal at the head of a production from attributes taken from the body of the production.
- An SDD without side effects is sometimes called an attribute grammar. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

Evaluating an SDD at the Nodes of a Parse Tree:

- The rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree.
- A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree. Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends.
- For example, if all attributes are synthesized, then we must evaluate the attributes at all of the children of a node before we can evaluate the attribute at the node itself.

Evaluation Orders for Syntax Directed Definitions:

- Dependency graphs are a useful tool for determining an evaluation order for the attribute instances in a given parse tree.
- While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.
- The two important classes of SDD are the "S-attributed" and "L-attributed" SDD's

Dependency Graphs:

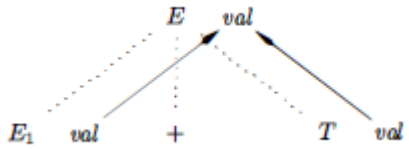
- A dependency graph depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second.
- Edges express constraints implied by the semantic rules.

Example:

Consider the following production and rule

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

- At every node N labeled E, with children corresponding to the body of this production, the synthesized attribute val at N is computed using the values of val at the two children, labeled E and T.
- As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.



Ordering the Evaluation of Attributes:

- The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of a parse tree. If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N.
- Thus, the only allowable orders of evaluation are those sequences of nodes N_1 , N_2,…, N_k such that if there is an edge of the dependency graph from N_i to N_j , then i<j. Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.

THREE ADDRESS CODE

- Three Address code is a sequence of statements of the general form

  x := y op z

  where x, y, z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed-or-floating point arithmetic operator, or a logical operator on Boolean-valued data.
- In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like x+y*z might be translated into the sequence of three-address instructions

  t1 := y * z

  t2 := x + t1

  where t1 and t2 are compiler-generated temporary names. The use of names for the intermediate values computed by a program allows three address code to be easily rearranged unlike postfix notation.

Implementations of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are

- Quadruples
- Triples
- Indirect triples

Quadruples:

- A quadruple is a record structure with four fields, which we call op, argl, arg2, and result. The op field contains an internal code for the operator.
- The three-address statement x:= y op z is represented by placing y in arg1, z in arg2 and x in result.
- Statements with unary operators like x: = – y or x: = y do not use arg2.
- Operators like param use neither arg2 nor result.
- Conditional and unconditional jumps put the target label in result.
- The quadruples for the assignment a: = b * –c + b * –c

$t_1 = \text{minus } c$
$t_2 = b * t_1$
$t_3 = \text{minus } c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

	op	Arg1	Arg2	Result
(0)	minus	c		t1
(1)	*	b	t1	t2
(2)	minus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

- The contents of fields arg1, arg2, and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

- To avoid entering temporary names into the symbol table. we might refer to a temporary value by the position of the statement that computes it.

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

	op	Arg1	Arg2
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

- In Triples, three-address statements can be represented by records with only three fields: op, arg1 and arg2. The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- In practice, the information needed to interpret the different kinds of entries in the arg1 and arg2 fields can be encoded into the op field or some additional fields.

Indirect Triples:

- Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves.
- This implementation is naturally called indirect triples. For example, let us use an array statement to list pointers to triples in the desired order.

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

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	Op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Quadruples Vs Triples Vs Indirect Triples:

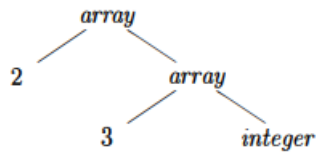
- A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around.
- With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change.
- With triples, the result of an operation is referred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

TYPES AND DECLARATIONS

- Type checking uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator.
- Translation Applications. From the type of a name, a compiler can determine the storage that will be needed for that name at run time.

Type Expressions:

- Types have structure, represented using type expressions: a type expression is either a basic type or is formed by applying an operator called a type constructor to a type expression. The sets of basic types and constructors depend on the language to be checked.
- The array type int[2][3] can be read as "array of 2 arrays of 3 integers each" and written as a type expression array(2, array(3, integer)). This type is represented by the tree. The operator array takes two parameters, a number and a type.



- A basic type is a type expression. Typical basic types for a language include boolean, char, integer, float, and void (denotes "the absence of a value.")
- A type name is a type expression.
- A type expression can be formed by applying the array type constructor to a number and a type expression.
- A record is a data structure with named fields. A type expression can be formed by applying the record type constructor to the field names and their types. Record types will be implemented by applying the constructor record to a symbol table containing entries for the fields.
- A type expression can be formed by using the type constructor → for function types. We write s→t for "function from type s to type t" Function types will be useful when type checking.
- If s and t are type expressions, then their Cartesian product s x t is a type expression. Products are introduced for completeness; they can be used to represent a list or tuple of types (e.g., for function parameters). We assume that x associates to the left and that it has higher precedence than→.
- Type expressions may contain variables whose values are type expressions.

Type Equivalence:

- When are two type expressions equivalent? Many type-checking rules have the form, "if two type expressions are equal then return a certain type else error."
- Potential ambiguities arise when names are given to type expressions and the names are then used in subsequent type expressions.
- The key issue is whether a name in a type expression stands for itself or whether it is an abbreviation for another type expression.

DECLARATIONS:

A simplified grammar that declares just one name at a time

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

- The above grammar deals with basic and array types. Nonterminal D generates a sequence of declarations. Nonterminal T generates basic, array, or record types.
- Nonterminal B generates one of the basic types int and float. Nonterminal C, for "component," generates strings of zero or more integers, each integer surrounded by brackets.
- An array type consists of a basic type specified by B, followed by array components specified by nonterminal C.
- A record type (the second production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

Sequences of Declarations:

- Languages such as C and Java allow all the declarations in a single procedure to be processed as a group. A variable offset, is used to keep track of the next available relative address.

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

- The translation scheme deals with a sequence of declarations of the form T id, where T generates a type. Before the first declaration is considered, offset is set to 0.
- As each new name x is seen, x is entered into the symbol table with its relative address set to the current value of offset, which is then incremented by the width of the type of x.
- The semantic action within the production D → T id; D1 creates a symbol table entry by executing top.put(id.lexeme, T.type, offset). Here top denotes the current symbol table.
- The method top.put creates a symbol table entry for id.lexeme, with type T.type and relative address offset in its data area.

TRANSLATION OF EXPRESSIONS

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

Operations with Expressions:

- The syntax-directed definition builds up the three-address code for an assignment statement S using attribute code for S and attributes addr and code for an expression E.

- Attributes S.code and E.code denote the three-address code for S and E, respectively. Attribute E.addr denotes the address that will hold the value of E.

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

- The last production E→id has the semantic rule which defines E.addr to point to the symbol-table entry for this instance of id. Let top denote the current symbol table.
- Function top.get retrieves the entry when it is applied to the string representation id.lexeme of this instance of id. E.code is set to the empty string.
- The semantic rules for E→El + E2 , generate code to compute the value of E from the values of El and E2. Values are computed into newly generated temporary names.
- If El is computed into E1.addr and E2 into E2.addr, then El + E2 translates into t= E1.addr + E2.addr, where t is a new temporary name. E. addr is set to t. A sequence of distinct temporary names t1, t2 , . . is created by successively executing new Temp( ).
- E.code is built by concatenating E1.code, E2.code and an instruction that adds the values of E1 and E2.

BACKPATCHING

- A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump.
- For example, the translation of the boolean expression B in if (B) S contains a jump, for when B is false, to the instruction following the code for S.
- In a one-pass translation, B must be translated before S is examined. What then is the target of the goto that jumps over the code for *S?*

One-Pass Code Generation Using Backpatching

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

  1. *makelist(i)* creates a new list containing only *i,* an index into the array of instructions; *makelist* returns a pointer to the newly created list.

  2. merge(pi,p2) concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.

  3. backpatch(p,i) inserts i as the target label for each of the instructions on the list pointed to by p.

Backpatching for Boolean Expressions

- We now construct a translation scheme suitable for generating code for boolean expressions during bottom-up parsing.
- A marker nonterminal *M* in the gram-mar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated. The grammar is as follows:

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

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

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

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

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

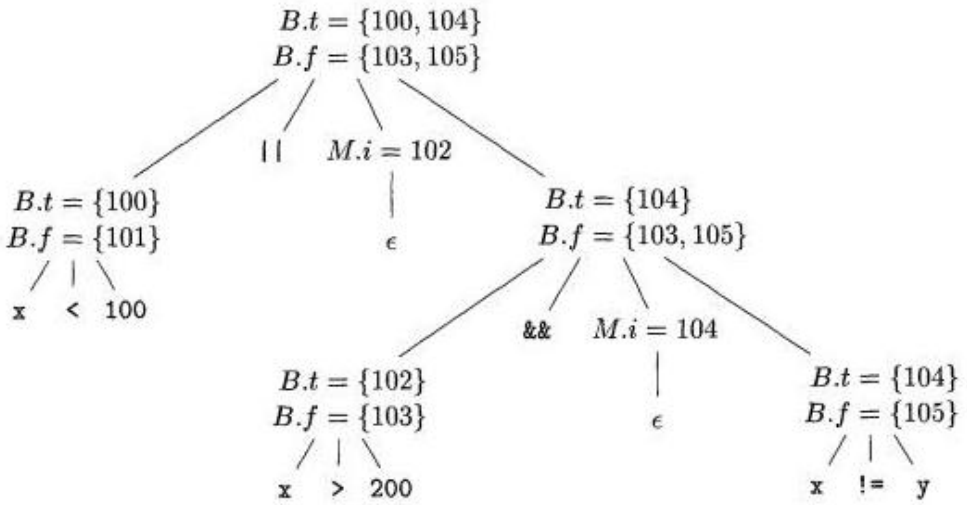
5) $B \rightarrow E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr)$;
$B.falselist = makelist(nextinstr + 1)$;
$emit('\texttt{if}' \ E_1.addr$ **rel**$.op \ E_2.addr \ '\texttt{goto} \ \_')$;
$emit('\texttt{goto} \ \_')$; }

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

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

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

- Consider semantic action for the production B -> B1 || M B2. If Bx is true, then B is also true, so the jumps on Bi.truelist become part of B.truelist.
- If Bi is false, however, we must next test B2, so the target for the jumps B>i .falselist must be the beginning of the code generated for B2.
- This target is obtained using the marker nonterminal M. That nonterminal produces, as a synthesized attribute M.instr, the index of the next instruction, just before B2 code starts being generated.
- The variable nextinstr holds the index of the next instruction to follow. This value will be backpatched onto the Bi.falselist (i.e., each instruction on the list B1.falselist will receive M.instr as its target label) when we have seen the remainder of the production B ->• B1 || M B2.



Flow-of-Control Statements

We now use backpatching to translate flow-of-control statements in one pass. Consider statements generated by the following grammar:

$$S \rightarrow \textbf{if}\,(B)\,S \mid \textbf{if}\,(B)\,S\,\textbf{else}\,S \mid \textbf{while}\,(B)\,S \mid \{\,L\,\} \mid A\;;$$
$$L \rightarrow L\,S \mid S$$

Here S denotes a statement, L a statement list, A an assignment-statement, and B a boolean expression. Note that there must be other productions, such as

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(a) After backpatching 104 into instruction 102.

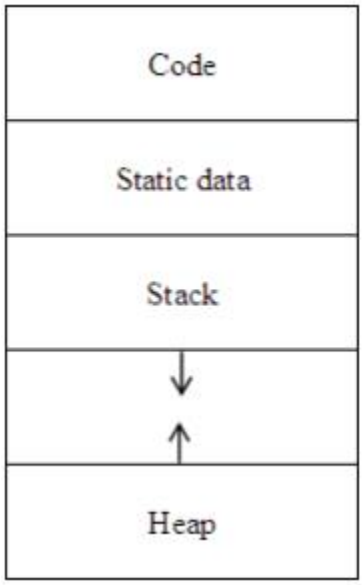
```
100:  if x < 100 goto _
101:  goto 102
102:  if y > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

(b) After backpatching 102 into instruction 101.

- The code layout for if-, if-else-, and while-statements is the same as in Section 6.6. We make the tacit assumption that the code sequence in the instruction array reflects the natural flow from one instruction to the next.

STORAGE ORGANIZATION

- From the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the compiler, operating system, and target machine. The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory.
- The run-time representation of an object program in the logical address space consists of data and program areas.



- The run-time storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory. Multi byte objects are stored in consecutive bytes and given the address of the first byte.
- The amount of storage needed for a name is determined from its data type, such as basic type character, integer, float or aggregate type like array, structure.
- The storage layout for data objects depends on the addressing constraints of the target machine. On many machines, instructions to add integers may expect integers to be aligned, that is, placed at an address divisible by 4.
- Although a character array (as in C) of length 10 needs only 10 bytes to hold ten characters, a compiler may allocate 12 bytes to get the proper alignment, leaving 2 bytes unused. Space left unused due to alignment considerations is referred to as padding.
- **CODE AREA**: The size of the generated target code is fixed at compile time, so the compiler places the executable target code in a statically determined area, the low end of memory.
- **STATIC AREA**: Size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area. In early versions of Fortran, all data objects could be allocated statically.
- **STACK AREA**: Used to store data structures called activation records that get generated during procedure calls.

- **HEAP AREA**: Many programming languages allow the programmer to allocate and deallocate data under program control. For example, C has the functions malloc and free that can be used to obtain and give back arbitrary chunks of storage. The heap is used to manage this kind of long-lived data.

Stack Allocation Space:

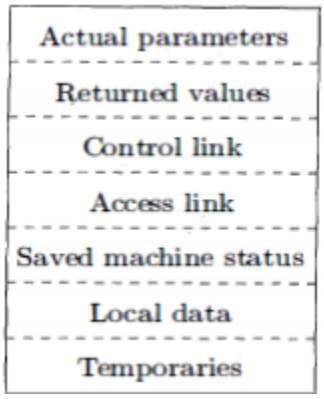
- Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped onto the stack.
- This arrangement not only allows space to be shared by procedure calls whose durations do not overlap in time, but it allows us to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same, regardless of the sequence of procedure calls.

Activation Trees:

- Stack allocation would not be feasible if procedure calls, or activations of procedures, did not nest in time. The following example illustrates nesting of Procedure calls.

Activation Records:

- Procedure calls and returns are usually managed by a run-time stack called the control stack. Each live activation has an activation record (sometimes called a frame) on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides. The latter activation has its record at the top of the stack.



1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.
2. Local data belonging to the procedure whose activation record this is.
3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the return address (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.
4. An “access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.
5. A control link, pointing to the activation record of the caller.
6. Space for the return value of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7. The actual parameters used by the calling procedure. Commonly, these Values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

Calling Sequences

- Procedure calls are implemented by what are known as calling sequences, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.
- The code in a calling sequence is often divided between the calling procedure (the "caller") and the procedure it calls (the "callee").
- In general, if a procedure is called from n different points, then the portion of the calling sequence assigned to the caller is generated n times. However, the portion assigned to the callee is generated only once.

ISSUES IN THE DESIGN OF A CODE GENERATOR

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

Input to code generator:

- The input to the code generation consists of the intermediate representation of the source program produced by front end, together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation. Intermediate representation can be:
- Linear representation such as postfix notation
- Three address representation such as quadruples, triples, indirect triples
- Virtual machine representation such as byte code and stack machine code
- Graphical representations such as syntax trees and DAG's.

The Target program:

- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code.
- The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.
- A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.
- In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.
- In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack.
- To achieve high performance the top of the stack is typically kept in registers. Stack-based machines almost disappeared because the stack organization was too limiting and required too many swap and copy operations.

Memory management:

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- Labels in three-address statements have to be converted to addresses of instructions.

For example,

j : goto i generates jump instruction as follows :

- if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated.
- if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.
- For example, every three-address statement of the form x = y + z, where x, y, and z are statically allocated, can be translated into the code sequence.

```
LD  R0, y       // R0 = y        (load y into register R0)
ADD R0, R0, z   // R0 = R0 + z   (add z to R0)
ST  x, R0       // x = R0        (store R0 into x)
```

- This strategy often produces redundant loads and stores. For example, the sequence of three-address statements.

```
a = b + c
d = a + e
```

- would be translated into

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

- Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if a is not subsequently used.
- The quality of the generated code is usually determined by its speed and size.

Register allocation:

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two subproblems:
- Register allocation - the set of variables that will reside in registers at a point in the program is selected.
- Register assignment - the specific register that a variable will reside in is picked.

Evaluation order:

- The order in which the computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.

BASIC BLOCKS

- Our first job is to partition a sequence of three-address instructions into basic blocks. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction.
- In the absence of jumps and labels, control proceeds sequentially from one instruction to the next. This idea is formalized in the following algorithm.

Algorithm

INPUT: A sequence of three-address instructions.

OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.

METHOD: First, we determine those instructions in the intermediate code that are leaders that is the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

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

- In generating the intermediate code, we have assumed that the real-valued array elements take 8bytes each, and that the matrix a is stored in row-major form.

Next-Use Information:

- Knowing when the value of a variable will be used next is essential for generating good code. If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.
- The use of a name in a three-address statement is defined as follows. Suppose three-address statement i assigns a value to x. If statement j has x as an operand, and control can ow from statement i to j along a

path that has no intervening assignments to x, then we say statement j uses the value of x computed at statement i. We further say that x is live at statement i.

Algorithm

INPUT: A basic block B of three-address statements. We assume that the symbol table initially shows all non-temporary variables in B as being live on exit.

OUTPUT: At each statement i: x = y + z in B, we attach to i the liveness and next-use information of x, y, and z.

METHOD: We start at the last statement in B and scan backwards to the beginning of B. At each statement i: x = y + z in B, we do the following:

1. Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, and z.
2. In the symbol table, set x to \not live" and \no next use."
3. In the symbol table, set y and z to \live" and the next uses of y and z to i

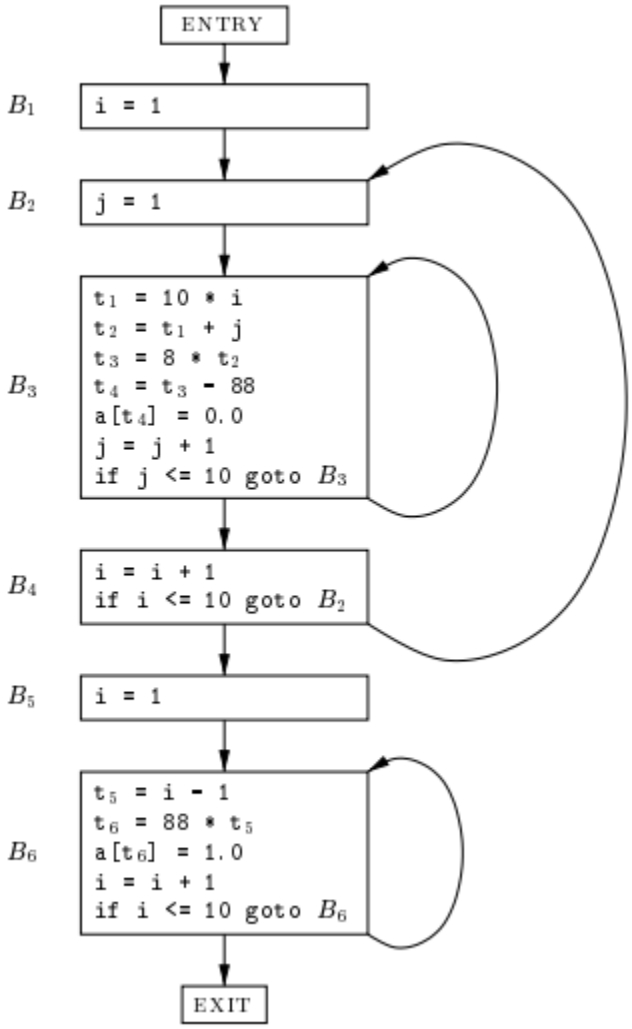
Here we have used + as a symbol representing any operator. If the three-address statement i is of the form x = + y or x = y, the steps are the same as above, ignoring z.

FLOW GRAPHS

- Once an intermediate-code program is partitioned into basic blocks, we represent the ow of control between them by a ow graph. The nodes of the flow graph are the basic blocks.
- There is an edge from block B to block C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B. There are two ways that such an edge could be justified:
- There is a conditional or unconditional jump from the end of B to the beginning of C.
- C immediately follows B in the original order of the three-address instructions, and B does not end in an unconditional jump.

Often, we add two nodes, called the entry and exit, that do not correspond to executable intermediate instructions. There is an edge from the entry to the first executable node of the ow graph, that is, to the basic block that comes from the first instruction of the intermediate code. There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program.

If the final instruction of the program is not an unconditional jump, then the block containing the final instruction of the program is one predecessor of the exit, but so is any basic block that has a jump to code that is not part of the program.



- The entry points to basic block B1, since B1 contains the first instruction of the program.
- The only successor of B1 is B2, because B1 does not end in an unconditional jump, and the leader of B2 immediately follows the end of B1.
- Block B3 has two successors. One is itself, because the leader of B3, instruction 3, is the target of the conditional jump at the end of B3, instruction 9.
- The other successor is B4, because control can fall through the conditional jump at the end of B3 and next enter the leader of B4.
- Only B6 points to the exit of the ow graph, since the only way to get to code that follows the program from which we constructed the ow graph is to fall through the conditional jump that ends B6.

Representation of flow graphs:

- Flow graphs, being quite ordinary graphs, can be represented by any of the data structures appropriate for graphs. The content of nodes (basic blocks) needs their own representation.
- We might represent the content of a node by a pointer to the leader in the array of three-address instructions, together with account of the number of instructions or a second pointer to the last instruction.

- However, since we may be changing the number of instructions in a basic block frequently, it is likely to be more efficient to create a linked list of instructions for each basic block.

Loops:

- Programming-language constructs like while-statements, do-while-statements, and for-statements naturally give rise to loops in programs.
- Since virtually every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops.
- Many code transformations depend upon the identification of “loops" in a ow graph. We say that a set of nodes L in a ow graph is a loop if L contains a node e called the loop entry, such that:
  1. e is not ENTRY, the entry of the entire ow graph.
  2. No node in L besides e has a predecessor outside L. That is, every path from ENTRY to any node in L goes through e.
  3. Every node in L has a nonempty path, completely within L, to e.

OPTICAL CODE GENERATION FOR EXPRESSIONS

- We can choose registers optimally when a basic block consists of a single expression evaluation, or if we accept that it is sufficient to generate code for a block one expression at a time.
- In the following algorithm, we introduce a numbering scheme for the nodes of an expression tree (a syntax tree for an expression) that allows us to generate optimal code for an expression tree when there is a fixed number of registers with which to evaluate the expression.

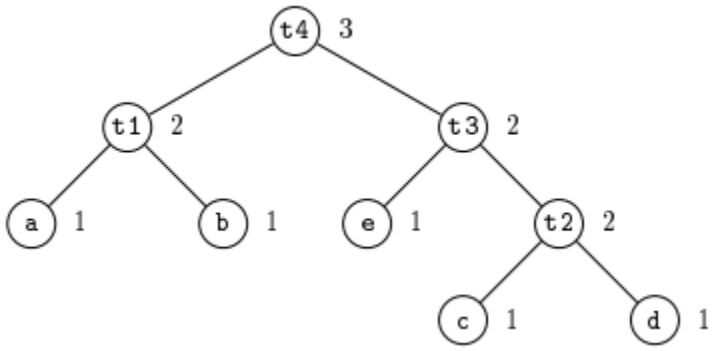
Ershov Numbers:

- We begin by assigning to each node of an expression tree a number that tells how many registers are needed to evaluate that node without storing any temporaries. These numbers are sometimes called Ershov numbers, after A.
- Ershov, who used a similar scheme for machines with a single arithmetic register.
- For our machine model, the rules are:
  1. Label all leaves 1.
  2. The label of an interior node with one child is the label of its child.
  3. The label of an interior node with two children is
     (a) The larger of the labels of its children, if those labels are different.
     (b) One plus the label of its children if the labels are the same.

Example: we see an expression tree (with operators omitted) that might be the tree for expression (a b) + e (c + d) or the three-address code

```
t1 = a - b
t2 = c + d
t3 = e * t2
t4 = t1 + t3
```

Each of the five leaves is labeled 1 by rule (1). Then, we can label the interior node for t1 = a - b, since both of its children are labeled. Rule (3b) applies, so it gets label one more than the labels of its children, that is, 2. The same holds for the interior node for t2 = c + d



Now, we can work on the node for t3 = e * t2. Its children have labels 1 and 2, so the label of the node for t3 is the maximum, 2, by rule (3a). Finally, the root, the node for t4 = t1 + t3, has two children with label 2, and therefore it gets label.

Generating code from labeled expression trees:

- It can be proved that, in our machine model, where all operands must be i registers, and registers can be used by both an operand and the result of an operation, the label of a node is the fewest registers with which the expression can be evaluated using no stores of temporary results.
- Since in this model, we are forced to load each operand, and we are forced to compute the result corresponding to each interior node, the only thing that can make the generate code inferior to the optimal code is if there are unnecessary stores of temporaries.
- The argument for this claim is embedded in the following algorithm for generating code with no stores of temporaries, using a number of registers equal to the label of the root.

Algorithm: Generating code from a labeled expression tree.

INPUT: A labeled tree with each operand appearing once (that is, no common subexpressions).

OUTPUT: An optimal sequence of machine instructions to evaluate the root into a register.

METHOD: The following is a recursive algorithm to generate the machine code. The steps below are applied, starting at the root of the tree. If the algorithm is applied to a node with label k, then only k registers will be used. However, there is a "base" $b \geq 1$ for the registers used so that the actual registers used are $R_b, R_{b+1}, \ldots, R_{b+k-1}$. The result always appears in R_{b+k-1}.

1. To generate machine code for an interior node with label k and two children with equal labels (which must be $k-1$) do the following:
   (a) Recursively generate code for the right child, using base $b+1$. The result of the right child appears in register R_{b+k-1}.
   (b) Recursively generate code for the left child, using base b; the result appears in R_{b+k-2}.
   (c) Generate the instruction `OP` $R_{b+k-1}, R_{b+k-2}, R_{b+k-1}$, where `OP` is the appropriate operation for the interior node in question.
2. Suppose we have an interior node with label k and children with unequal labels. Then one of the children, which we'll call the "big" child, has label k, and the other child, the "little" child, has some label $m < k$. Do the following to generate code for this interior node, using base b:
   (a) Recursively generate code for the big child, using base b; the result appears in register R_{b+k-1}.
   (b) Recursively generate code for the little child, using base b; the result appears in register R_{b+m-1}. Note that since $m < k$, neither R_{b+k-1} nor any higher-numbered register is used.
   (c) Generate the instruction `OP` $R_{b+k-1}, R_{b+m-1}, R_{b+k-1}$ or the instruction `OP` $R_{b+k-1}, R_{b+k-1}, R_{b+m-1}$, depending on whether the big child is the right or left child, respectively.
3. For a leaf representing operand x, if the base is b generate the instruction `LD` R_b, x.

PRINCIPAL SOURCES OF OPTIMIZATION

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global. Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations:

There are a number of ways in which a compiler can improve a program without changing the function it computes. Function preserving transformations examples:

- Common sub expression elimination
- Copy propagation
- Dead-code elimination
- Constant folding

Common Sub expressions elimination:

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

For example:

t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5

The above code can be optimized using the common sub-expression elimination as

t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5

The common sub expression t4: =4*i is eliminated as its computation is already in t1 and the value of i is not been changed from definition to use.

Copy Propagation:

Assignments of the form f: =g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

Example:

x=Pi;
……
A=x*r*r;

The optimization using copy propagation can be done as follows: A=Pi*r*r; Here the variable x is eliminated

Dead-Code Eliminations:

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

Example:

```
i=0;
if(i=1)
{
a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

Constant folding:

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding. One advantage of copy propagation is that it often turns the copy statement into dead code. For example,

a=3.14157/2 can be replaced by

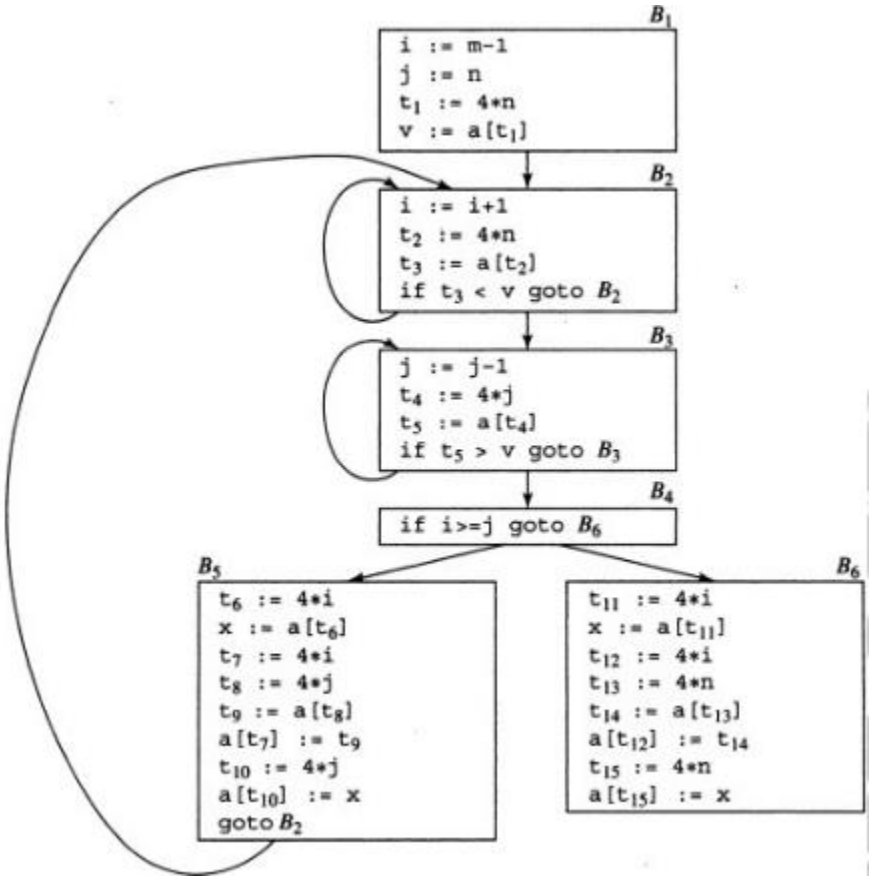
a=1.570 thereby eliminating a division operation.

Loop Optimizations:

In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of instructions in an inner loop is decreased, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- Code motion, which moves code outside a loop;
- Induction-variable elimination, which we apply to replace variables from inner loop.
- Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.



Code Motion:

An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

while (i <= limit-2) /* statement does not change limit*/ Code

motion will result in the equivalent of

t= limit-2;

while(i<=t) /* statement does not change limit or t*/

Induction Variables:

Loops are usually processed inside out. For example, consider the loop around B3. Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables. When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig.5.3 we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction variable elimination. Eventually j will be eliminated when the outer loop of B2- B5 is considered.

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift.

PEEP-HOLE OPTIMIZATION

Most production compilers produce good code through careful instruction selection and register allocation, a few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program.

A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.

Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

Characteristic of peephole optimization:

- Each improvement may spawn opportunities for additional improvements.
- Repeated passes over the target code are necessary to get the maximum benefit.

Examples of program transformations those are characteristic of peephole optimizations:

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Eliminating Redundant Loads and Stores

If we see the instruction sequence in a target program,

```
LD a , R0
ST R0 , a
```

We can delete store instructions because whenever it is executed. First instruction will ensure that the value of has already been loaded into register R0.

Eliminating Unreachable Code

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabelled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions.

For example, for debugging purposes, a large program may have within it certain code fragments that are executed only if a variable debug is equal to 1.

In C, the source code might look like:

```
#define debug 0
....

if ( debug ) {
Print debugging information

}
```

In the intermediate representation, this code may look like

```
if debug == 1 goto L1
goto L2
L 1 : print debugging information
L2 :
```

Flow-of-Control Optimizations

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the sequence

```
goto L1
L1 : goto L2
```

by the sequence

```
goto L2
L1 : goto L2
```

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

Similarly, the sequence

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

can be replaced by the sequence

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

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

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

may be replaced by the sequence

```
L3 :
if a < b goto L2
goto L3
```

Algebraic Simplification and Reduction in Strength

These algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as

```
x = x + 0
      or
x= x * 1
```

in the peephole.

Similarly, reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

Use of Machine Idioms

• The target machine may have hardware instructions to implement certain specific operations efficiently.

• Detecting situations that permit the use of these instructions can reduce execution time significantly.

For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.

The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like x = x + 1.

x = x + 1 → x++

x = x - 1 → x- -

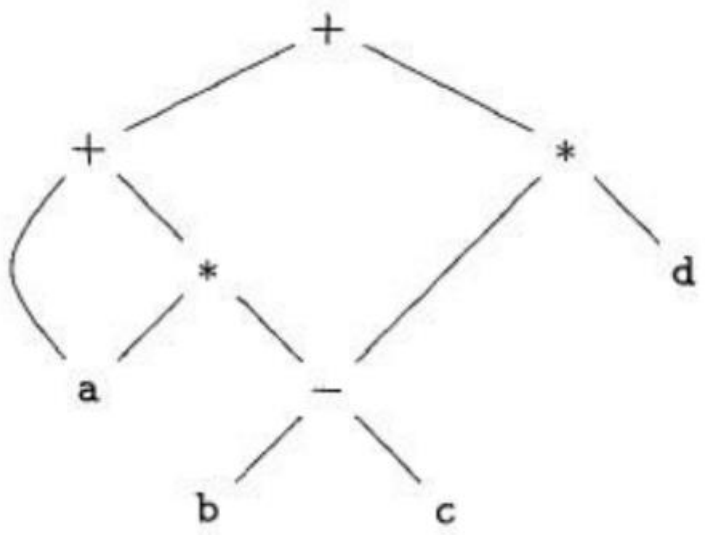
OPTIMIZATION OF BASIC BLOCKS

We can often obtain a substantial improvement in the running time of code merely by performing local optimization within each basic block by itself.

DIRECTED ACYCLIC GRAPHS (DAG)

- Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators.
- The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression.
- Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Example: The DAG for the expression a+a*(b-c)+(b-c)*d by sequence of steps The leaf for "a" has two parents, because a appears twice in the expression. More interestingly, the two occurrences of the common subexpression b-c are represented by one node, the node labeled "-". That node has two parents, representing its two uses in the subexpressions a*(b-c) and (b-c)*d. Even though b and c appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression b-c.



The DAG Representation of Basic Blocks

DAG for a basic block as follows:

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.

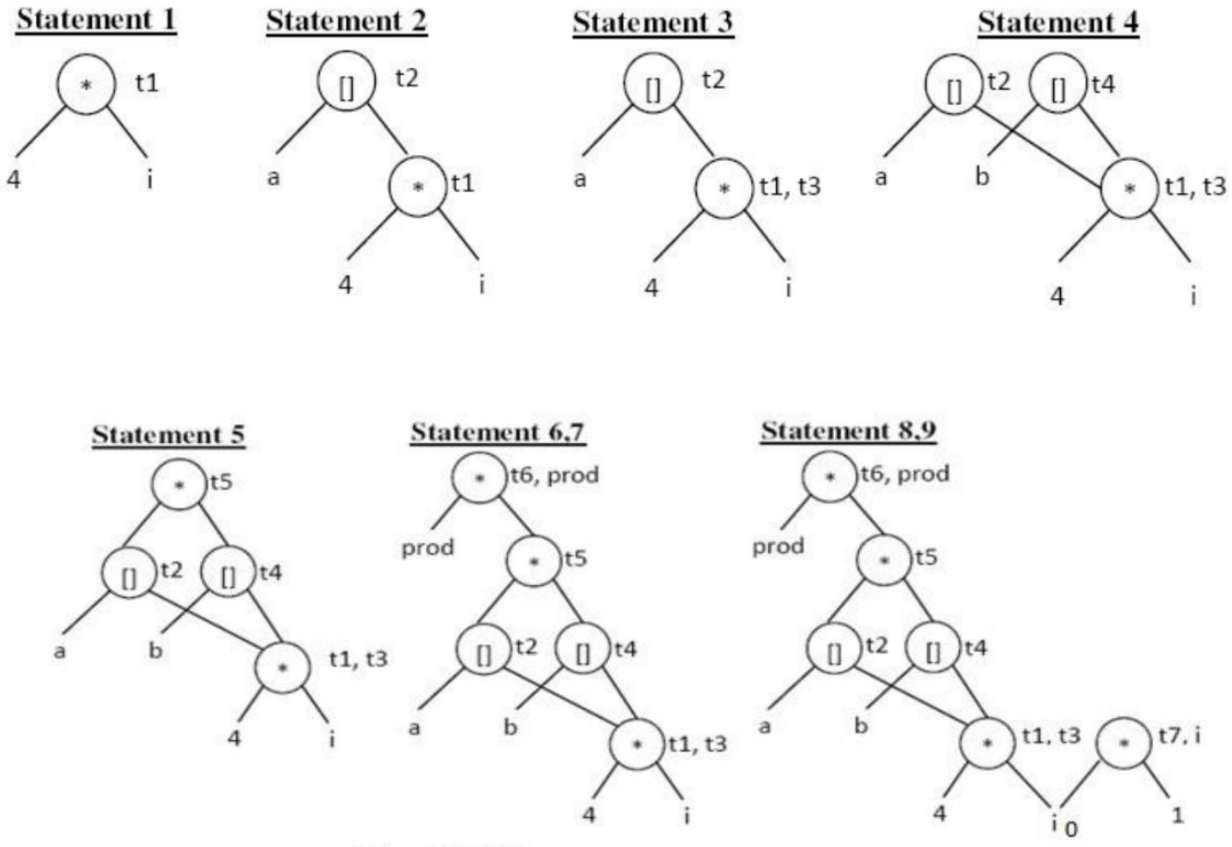
2. There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.

3. Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block.

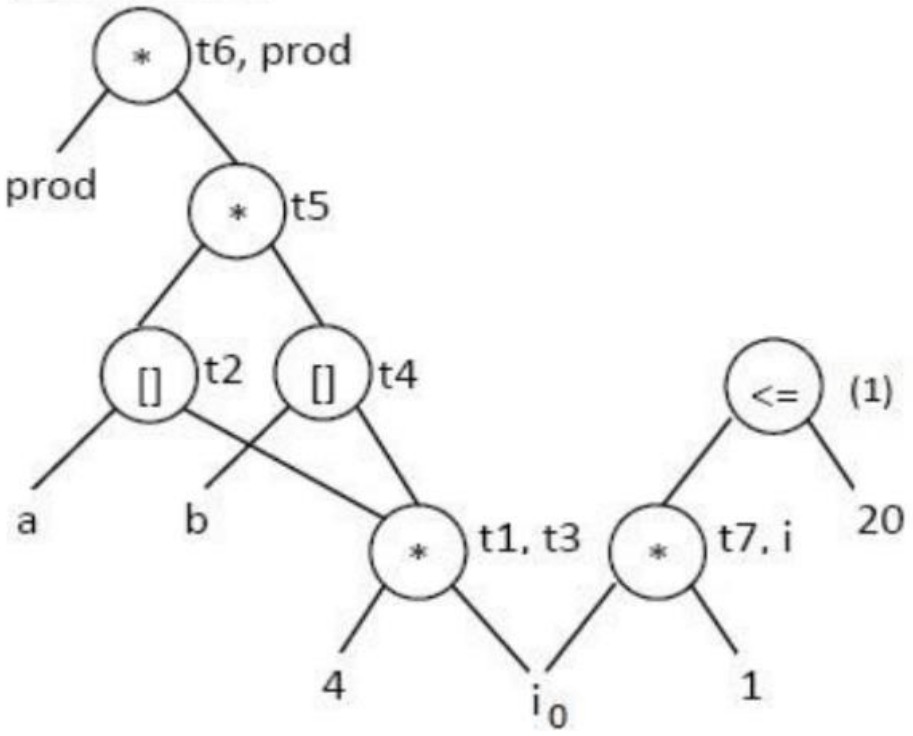
4. Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block; that is, their values may be used later, in another block of the flow graph. Calculation of these "live variables" is a matter for global flow analysis, The DAG representation of a basic block lets us perform several code improving transformations on the code represented by the block.

Example: Construct DAG from the basic block.

```
1 t1 = 4*i
2 t2 = a[t1]
3 t3 = 4*i
4 t4 = b[t3]
5 t5 = t2*t4
6 t6 = prod + t5
7 t7 = i+1
8 i = t7
9 if i<=20 goto 1
```



Final DAG



GLOBAL DATA FLOW ANALYSIS

Data-flow information can be collected by setting up and solving systems of equations that relate information at various points in a program. A typical equation has the form

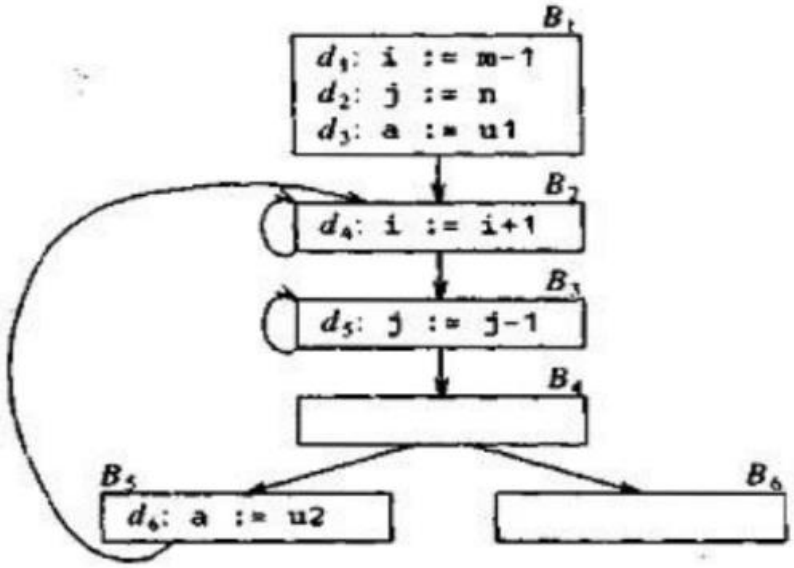
$$out[S] = gen[S] \cup (in[S] - kill[S])$$

The information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement. Such equations are called data-flow equations. The details of how data-flow equations are set up and solved depend on three factors.

1. The notions of generating and killing depend on the desired information. i.e., on the data-flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with the flow of control and defining out[S] in terms of in[S], we need to proceed backwards and define in[S] in terms of out[S].
2. Since data flows along control paths, data-flow analysis is affected by the control constructs in a program. In fact, when we write out[S] we implicitly assume that there is unique end point where control leaves the statement.
3. There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths

Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.



Now, let us take a global view and consider all the points in all the blocks. A path from P1 to Pn is a sequence of points P1, P2, . . . , Pn such that for each I between 1 and n-1, either

1. Pi is the point immediately preceding a statement and Pi+1 is the point immediately following that statement in the same block, or
2. Pi is the end of some block and Pi+1 is the beginning of a successor block.

Reaching Definitions

A definition of a variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an I/O device and store it in x. These statements certainly define a value for x, and they are referred to as unambiguous definitions of x.

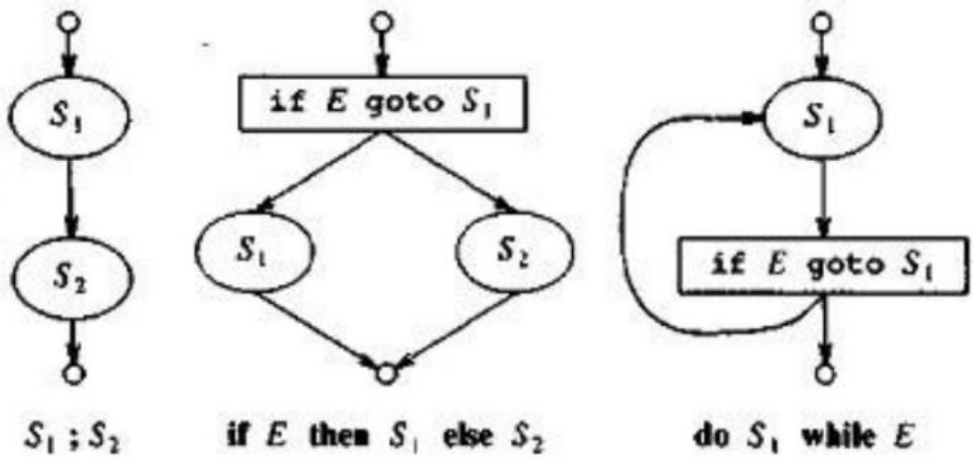
There are certain other kinds of statements that may define a value for x; they are called ambiguous definitions. The most usual forms of ambiguous definitions or x are:

1. A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure. We also have to consider the possibility of "aliasing;" where x is not in the scope of the procedure, but x has been identified with another variable that is passed as a parameter or is in the scope.
2. An assignment through a pointer that could refer to x. For example, the assignment *q:=y is a definition of x if it is possible that q points to x.

Data-Flow Analysis of Structured Programs

Consider the production

S -> id :=E | S;S | if E then S else S | do S while E E -> id + id | id



We define a portion or a flow graph called a region to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.

Representation of Sets

Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then the bit vector representing a set of definitions will have 1 in positions i if and only if the definition numbered i is in the set.

A bit-vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A^ ¬ B.