Compilers
-- Basic functions

- Generally, an independent course, maybe plus one semester on implementation of a compiler.
- High level language program → program in assembly language or object code directly
- Three steps:
  - Scanning, parsing, and (object) code generation.
  - (more general, five steps: scanning, parsing, intermediate code generation, code optimization, and object code generation)
- What does a program consists of?
  - a string of characters? From lowest level, yes.
  - A sequence of tokens: a keyword, a variable name, an integer, an operator, …,
    - each token consists of a string of characters satisfying some rules/format, called lexical rules.
  - A sequence of statements, such as a declaration statement, an assignment statement, an IF statement.
    - Each statement consists of tokens satisfying some rules/format, called syntax or grammar.
    - Each statement also has a specific meaning, called semantics.

Compilers
-- Basic functions (cont.)

- Scanning:
  - Scan the character string of a program, analyze them (according to rules, called lexical rules), then figure out each token. Also called lexical analysis. The part of a compiler for this function is called scanner.
- Parsing:
  - Pass through the sequence of tokens, parse them (according to rules, called grammars), then figure out each statement, also called syntactic analysis. The part of a compiler for this function is called parser.
- (Object) code generation:
  - Each statement has its meaning (semantics), for each parsed statement, generate its code according to its meaning. Also called semantic analysis. The part of a compiler for this function is called code-generator.
- Notes:
  - Three steps, not necessarily three passes.
  - Comparison with assembly language program and assemble:
    - What does an assembly language program consists of? How does the assembler finish its task?
- Example of a Pascal program (Figure 5.1)

Grammars

- Syntax and semantics of a statement.
- Grammar:
  - Describe the form, or syntax, of the legal statements in high level language.
  - Does not describe the meaning of a statement.
  - For examples:
    - I:=J+K and X:=Y+I
    - Where I, J, K are INTEGER variables and X,Y are REAL variables.
    - They have identical syntax: an assignment, the value to be assigned is given by a two variable expression with operator +.
    - Recall SIX machine: integer arithmetic and floating-point arithmetic.
    - The first assignment will add J and K together and store to I.
    - The second assignment needs to first convert I to floating point and add two floating-point numbers and then store it to X.
    - So they will generate different machine codes.
  - The meaning of a statement is used in or to say, controlled by code-generation.
- Many different ways to describe grammars.
  - One typical way is BNF (Backus-Naur Form). We will use it.

Grammars (cont.)

- BNF grammars: consists of a set of rules, each of which defines the syntax of some construct in the programming language.
  - <read>::= READ(<id-list>)
  - ::= : is defined as
  - The left symbol: a language construct being defined
  - The right side: description of the syntax being defined for it.
  - A name between < and > is a nonterminal symbol, which is the construct of the language
  - Entities not in < and > are terminal symbols, also called tokens.
  - In the example, <read> and <id-list> are non-terminal symbols, and READ, (, and) are terminal symbols.
  - It says that language construct <read> consists of tokens READ, followed by token (, followed by a language construct <id-list>, and followed by token).
  - Note: blanks are not significant and they are there simply for readability.
  - Furthermore, <id-list>::==id | <id-list>, id
    - Recursive definition. id is an identifier that is recognized by the scanner.
    - So <id-list> consists of one or more id's separated by commas.
    - ALPHA is an <id-list>. ALPHA, BETA is another <id-list>
    - More examples: READ(VALUE), <assign>::= id:=<exp>, <exp>::=<term>| <exp> + <term>| <exp> - <term>
- Simplified Pascal grammars (Figure 5.2)
- Parse tree for two statements in Fig. 5.1 (Figure 5.3)
- Parse tree for the entire program in Figure 5.1 (Figure 5.4) and Figure5.4 (cont'd)

Lexical analysis

- Scan the program to be compiled and recognize the tokens (from string of characters).
- Tokens make up the source program.
- Tokens: keywords, operators, and identifiers, integers, floating-point numbers, character strings, and other items.
- Token coding scheme for the grammar from Figure 5.2 (Figure 5.5)
- Lexical scan of the program from Fig5.1 (Figure 5.6)
  - Line, token type, token specifier
  - Where token specifier is either a pointer to a symbol table or the value of integers.
- Relation among scanner and parser
  - As a subroutine, called by parser, and return a token on each call.
  - Scanner reads line by line of the program.
  - Comments are ignored by scanner.
  - Possibly print the source listing.
- Specific features of different languages
  - Any specific format of statements, such as FORTRAN. Column 1-5: line number, not integer.
  - Whether blanks as delimiters for tokens (such as PASCAL) or not (such as FORTRAN).
  - Continuation of one line to the next line (as in PASCAL) or continuation flag (as in FORTRAN).
  - Rules for token formation, such as READ within a quoted character string, may not be a token.
  - MOREOVER, DO 10 I=1,100 and DO 10 I=1.
  - Languages without reserved key words: IF, THEN and ELSE may be keyword or variable names:
    - IF (THEN .EQ. ELSE) THEN IF=THEN ELSE THEN=IF ENDIF is legal in FORTRAN.

Lexical analysis
--Modeling scanners as finite automata

- A set of *state*s and a set of *transition*s from one state to another. One initial state and some final states.
  - Begin at start state, scan the characters and transit to next states, and recognize a token when reaching a final state.
- Graphical representation of a finite automaton (Figure 5.7)
- Finite automata for typical programming language tokens (Figure 5.8)
  - (a) recognizes identifiers and keywords beginning with letters and then letters and/or digits.
  - (b) allows _ in identifiers and keywords.
  - (c) recognizes integers, allowing leading 0.
  - (d) does not allow leading 0, except number 0.
- Finite automaton to recognize tokens from Fig 5.5 (Figure 5.9)
  - Same final state for keywords and identifiers, so a special keyword look-up is needed for distinguishing keywords.
  - In addition, a special check for the length of identifiers, if any. (Automata can not represent the limitation on the length of strings.
  - State 3 is for keyword END., should check for others such as VAR., in this case, to move back to state 2.
- Finite automata provides an easy way to visualize the operation of scanner. However, the real advantage is its easy implementation.
- Token recognition using (a) algorithmic code and (b) tabular representation of finite automaton (Figure 5.10)
  - The tabular implementation is more preferred.

Syntactic Analysis

- The source statements written by programmers are recognized as language constructs described by the grammar.
  - Building the parse tree for the statements being translated.
- Bottom-up and top-down techniques.
  - Bottom-up: building the leave of the tree first which match the statements, and then combining into higher-level nodes until the root is reached.
  - Top-down: beginning from the root, i.e., the rule of the grammar specifying the goal of the analysis, and constructing the tree so that the leave match the statements being analyzed.

Bottom-up parsing --operator-precedence parsing

- E.g. A+B*C-D,
  - + < * and * > -,
  - i.e. multiplication and division have higher precedence than addition and subtraction.
  - So when analyzing the above expression, B*C will be constructed first at the lower level and then + and – at the higher level.
  - The statement being analyzed is scanned for a sub-expression whose operator has higher precedence than the surrounding operators.
- Here, operator is taken to mean any terminal symbol (i.e. , any token), so there are precedence relation among BEGIN, END, id, (, ...
  - E.g., PROGRAM = VAR, and BEGIN < FOR
  - also, ; >END as well as END >;
  - Note: there is no relation between some pairs of tokens, which means that the second token should not follow the first one. If this occurs, it is an error.
  - If there is a relation, it is unique. Non-unique relation will make operator-precedence parsing fail.
- Precedence matrix for the grammar from Fig 5.2 (Figure 5.11)
  - (Automatically) Generated from the grammar in Figure 5.2.
- Operator-precedence parse of a READ statement from line 9 of program 5.1 (Figure 5.12)
  - BEGIN < READ, so keep BEGIN (in stack) and proceed to READ.
  - READ = (, so they are children of a same parent.
  - (< id, so keep READ and (and proceed to id, VALUE is recognized as id (from Scanner).
  - id>), id is recognized as <id-list>, simply denoted as <N1> (a non-terminal). If follow rule 12 of the grammar, id can be recognized as <factor>, but it does not matter.
  - (=), so READ, (, <N1>,) are all the children of the same parent.
  -)>;, finishing the parsing tree for READ, by following rule 13, get non-terminal <read>, simply denoted as <N2>.
  - Please compare this parsing tree with Fig. 5.3 (a), they are same except the name of non-terminals.

Bottom-up parsing (cont.) --operator-precedence parsing

- Operator-precedence parse of an assignment in line 14 of program 5.1 (Figure 5.13) and Figure 5.13 (cont'd)
  - Note: the left-to-right scan is continued in each step only far enough to determine the next portion of the statement to be recognized, i.e., the first portion delimited by < and >.
  - Each portion is constructed bottom-up.
  - Compare this parsing tree with Fig5.3(b). In Fig5.3(b), id SUMSQ →<factor> →<term> which is one operand of DIV. But here, SUMSQ is interpreted as <N1>. These differences are consistent with the use of arbitrary names for non-terminals. The id, <factor> <term> grammar are for determining the precedence, which is combined in our parsing precedence matrix, so, no need in the analyzed parsing tree.
- Work on the entire program Fig 5.1, following the operator-precedence parsing method, the resulting parsing tree similar to Fig 5.4 will be generated, except for the differences in the naming of non-terminals.
- Other parsing techniques:
  - Shift-reduce parsing technique:
    - Shift: pushing the current token into stack, similar to when < and = are encountered
    - Reduce: recognize symbols on top of the stack according to a rule of the grammar, similar to when > is encountered.
    - Example of shift-reduce parsing (Figure 5.14)
  - LR(k) parsing technique: the most powerful one.
    - K: indicates the number of tokens following the current position that are considered in making parsing decision.
  - SLR and LALR techniques, for more restricted set of grammars.

Top-down parsing technique --Recursive-descent parsing

- For each non-terminal, there is a procedure which
  - Begins from the current token, search the following tokens, and try to recognize the rule associated with the non-terminal.
  - May call other procedures or even itself for non-terminals included in the rule of this non-terminal.
  - When a match is recognized, the procedure returns an indication of success, otherwise, error.
- Recursive-descent parse of a READ statement (Figure 5.16)
  - Begins from token READ, then search for next token (, then call procedure IDLIST, and after IDLIST returns true, search for token), if success, return true (and goes to next token), otherwise, return false.
- There are several alternatives for a non-terminal:
  - The procedure must determine which alternative to try.
  - For recursive-descent parsing, assume that the next input token is enough to check for making the decision.
    - E.g., for the procedure of <stmt>, see the next token, if READ, then call <read>, if id, then call <assign>, if FOR, call <for>, and if WRITE, call <write>.
- Problem with left recursive non-terminal such as <id-list>
  - If try <id-list>,id, then recursively call itself, and call itself again, so unending chain without consuming any input token.
  - So top-down parsers cannot be directly used with a grammar containing left-recursive rules.
- Solution:
  - Change the rule <id-list>::=id|<id-list>,id to <id-list>::=id {,id}, where {,id} means that the zero or more occurrences of ,id.
  - This kind of rule is an extension to BNF.
  - Similar for others: such as: <exp>::=<term> {+ <term> | - <term>}
  - Simplified Pascal grammar modified for recursive-descent parse (Figure 5.15)
    - Note: <exp> → <term> → <factor> → (<exp>). This kind of indirect recursive call is OK. Since this chain of calls will consume at least one token from the input. That is the progress can be made.
  - Please look at IDLIST procedure in above Figure 5.16 for modified <id-list> rule.

Top-down parsing technique (cont.) --Recursive-descent parsing

- Graphical representation of the recursive-descent parsing process for READ statement (Figure 5.16 cont'd)
  - Part (iii) is same as the parse tree in Figure 5.3(a).
  - Here tree is constructed in a top-down manner.
- Recursive-descent parsing procedures of an assignment statement (Figure 5.17) and Figure 5.17 (Cont'd)
- Graphical representation of recursive-descent parsing of assignment statement (Figure 5.17 cont'd) and Figure 5.17 (Cont'd)
  - Compare the parsing tree in Figure 5.17 (b) to the one in Figure 5.3(b). They are nearly similar. The differences correspond to the differences between grammars in Figure 5.15 and Figure 5.2.
- You can write all the procedures for the entire grammars, beginning the parsing by calling procedure <prog>. The top-down parsing tree will be similar to the one in Figure 5.4.
- Combination of top-down and bottom-up techniques:
  - Recursive-descent parsing for high-level constructs and operator-precedence parsing for expressions.

Code generation

- Generate object code in the form of machine code directly or assembly language.
- A basic technique:
  - Associate each rule (or an alternative rule) of the grammar with a routine, which translates the construct into object code according to its meaning/semantics.
  - Called semantic routine or code-generation routine.
  - Possibly generate an intermediate form so that optimization can be done to get more efficient code.
  - Code generation needs not to be associated with a specific parsing technique.
  - Code generated clearly depends on a particular machine. Here assume SIC/XE.
- Data structures needed:
  - A list, or a queue, first-in-first-out, also a LISTCOUNT variable
  - A stack, first-in-last-out.
  - S(token): specifier of a token, i.e., a pointer to the symbol table or the integer value.
  - In addition, S(<non-terminal>) for a non-terminal, is set to rA, indicating that the result of code for this construct is stored in register A.
  - LOCCTR: location counter, indicating the next available address.
- The generated code here is represented in SIC assembly language, for easy understanding and readability.
- Code generation for a READ statement (Figure 5.18)
  - (a): parsing tree, no matter what parsing technique is used, the left-most substring which can be interpreted by a rule of the grammar is recognized:
    - In recursive-descent, a recognition occurs when a procedure returns to its caller, with a success.
    - In operator-precedence, a recognition occurs when a substring of input is reduced to a non-terminal.
    - Thus, the parse first recognizes the id VALUE as an <id-list>, and then recognizes the complete statement as <read>.
  - (b): A set of routines for the rules of this part.
  - (c): the symbolic representation of code generated.
    - +JSUB XREAD, call standard library function XREAD.
    - WORD 1, a word in memory, indicates how many data followed. WORD VALUE, a word in memory, indicates the address to store the read value.
    - The address of WORD 1 will be stored in register L when JSUB is executed, so that XREAD can access the memory following JSUB and put the read value in the address of WORD VALUE. Also after finishing, the XREAD will return to (L)+2.

Code generation (cont.)

- Code generation for an assignment statement (Figure 5.19) and Figure 5.19 (Cont'd) and Figure 5.19 (Cont'd)
  - The order in which the parts of the statement are recognized is the same as the order in which the calculations are performed.
  - As each portion of the statement is recognized, the corresponding routine is called to generate code.
  - E.g., <term>$_1$::=<term>$_2$*<factor>, all arithmetic operations are performed in register A.
    - Clearly need to generate MUL instruction. The result will be in A.
    - If either the result of <term>$_2$ or <factor> is already in A, (perhaps as the result of a previous computation), MUL is all we need.
    - Otherwise, we must generate LDA to load <term>$_2$ or <factor> to register A, preceding MUL. We may also need to save the previous value in A for later use (so generate STA instruction).
  - Need to keep track of the result left in A by each segment of code generated:
    - Specifier S(<non-terminal>) is set to rA, indicating that the result of this computation is in A
    - REGA: indicate the highest-level node of the parsing tree whose value is left in A by code generation so far. Clearly, just one such node at any time.
    - temporary variables introduced by compiler, which will be assigned memory locations in object code.
  - Similar code generation process for +, DIV and – operations.
  - For <assign>, it generates LDA <exp>, STA S(id), and REGA=NULL.
  - Rules in Figure 5.19 (b) do not generate object code, but set the node specifier of the higher-level node to reflect the location of the corresponding value.

Code generation (Cont'd)

- Other code-generation routines for the grammar in Figure 5.2 (Figure 5.20) and Figure 5.20 (Cont'd)
  - <prog-name> generates header information, similar to that created by START and EXTREF, it also generates save return address and jump to the first executable instruction.
  - When <prog> is recognized, storage locations are assigned to any temporary variables and any references to these variables are then fixed using the same process performed for forward references by one-pass assembler.
  - Regard <for>, a little more complicate.
    - When <index-exp> is recognized, code is generated to initialize the index variable and test for loop termination. The information is also stored in stack for later use.
    - For each statement in the body, code is generated
    - When the complete <for> is recognized, code is generated to increase the value of index variable and jump back to the beginning of the loop to test for loop termination. The information about the index variable, the beginning address of the loop, and the address of jumping out of loop is obtained from stack.
    - One feature is that stack is used and it will also allow nested for.
- Symbolic representation of object code generated for the program in Figure 5.1 (Figure 5.21)
  - Please go through it carefully and understand the compilation process completely.
  - Note: object code is in symbolic representation. Also code and data interleave.

Machine-Dependent Compiler Features

- In generally, high languages are machine-independent.
- But the generation and optimization of code is machine dependent.
- In order for code optimization, intermediate form of the program is usually generated.
- So: lexical analysis → syntactical analysis → intermediate form generation → optimization →code generation.
- Here discuss:
  - intermediate form generation (which is machine-independent, but needed for optimization)
  - Optimization (machined dependent).
  - Many machine independent optimizations will be discussed later.

Intermediate form of program

- Quadruples: operation, op1, op2, result.
- Ex1: SUM := SUM + VALUE
  - +, SUM, VALUE, i_1
  - :=, i_1, , SUM (:= is treated as an operator)
- EX2: VARIANCE := SUMSQ DIV 100 – MEAN*MEAN
  - DIV, SUMSQ, #100, i_1
  - *, MEAN, MEAN, i_2
  - -, i_1, i_2, i_3
  - :=, i_3, , VARIANCE
- Examples of optimization:
  - Re-arrange quadruples to remove redundant LOAD and STORE
  - Intermediate result is assigned to register to make code more efficient.
- Intermediate code for the program from Fig5.1 (Figure 5.22)
- How to generate intermediate form of the program?
  - Replace the code generation routines with intermediate form generation routines.

Machine-dependent optimization

- Assignment and use of registers
  - Faster than from memory
  - Try to have values in registers and use them as much as possible.
  - Ex: VALUE is used in 7 and then in 9. If enough registers are available, VALUE should be retained in a register.
  - Ex: in 16, i_5 is assigned to MEAN. If i_5 is assigned to a register, it can be used directly in 18.
  - Ex: machine code in Fig 5.21 uses only one register A to efficiently handle six of eight intermediate results (i_j) in Fig 5.22.
- Since the number of registers is fixed, how to select a register to replace when needed?
  - Scan the next point where a register's value will be used, the one with longest next point is selected for replacement.
  - The value in the replacement register may need to be stored in memory if not yet. (see GETA procedure which stores A's value to memory before load if needed.)
- Also need to consider control flow of a program, since JUMP creates difficulty to keep track of register contents.
    - Ex: Quadruple 1 assigns 0 to SUM, quadruple 7 uses SUM. If SUM is in a register, then use the register directly. However, the JUMP in quadruple 14 jumps to 4, then to 7. But this time, SUM may not be in the register.
    - Solution: divide a program into basic blocks with jump between blocks: each block has one entry point, one exit point (can be a jump), and no jump within the block.
      - Basic blocks and flow graph for the quadruples in Fig5.22 (Figure 5.23)
      - Then limit the register assignments to basic blocks.
      - More sophisticated techniques may allow register assignments from one block to the next.

Machine-dependent optimization (cont.)

- Re-arrangement of quadruples:
  - Rearrangement of quadruples for code optimization (Figure 5.24)
    - Reduce to 7 instruction from 9 after re-arrangement of quadruples.
- Other optimizations by taking advantages of specific machine characteristics and instructions:
  - Specific loop control or addressing models.
  - Calling procedures and manipulating data structures in a single operation.
  - Parallel features, if any.

Machine independent compiler features --structured variables

- Array, record, string, set,…
- A: array[1..10] of integer
  - If each integer is one word, then allocate 10 words for A.
  - Generally, array[l..u] of integer, allocate u-l+1 words.
- B: array[0..3,1..6] of integer, allocate 4*6=24 words.
  - Generally, array[$l_1..u_1,l_2..u_2$] of integer allocates $(u_1-l_1+1)*(u_2-l_2+1)$ words.
- Reference to an element of an array:
  - One dimension: direct correspondence
  - Multiple dimensions: row-major or column-major. FORTRAN uses column-major.
    - Storage of B: array[0..3,1..6] in (a) row-major order and (b) column-major order (Figure 5.25)
  - Must calculate the address of the element relative to the beginning of the array.
    - Such code is generated by compiler, put in the index register, use index addressing.
    - Ex. Reference to A[6], so the relative address is: 3*5=15. (3 bytes * 5 preceding elements). 15 can be computed by compiler.
    - However, reference to A[i], generate code to compute relative address: w*(i-l).
    - Moreover, reference to B[i,j], generate code to compute relative address: $w*[(i-l_1)*(u_2-l_2+1)+(j-l_2)]$
    - Code generation for array references (Figure 5.26)
  - The symbol table for array contains array types, dimensions, lower and upper limits.
    - It is enough for compiler to generate code if the information is constant/static, called static array.
  - Dynamic array:
    - INTEGER, ALLOCATABLE, ARRAY(: , :)::MATRIX, then ALLOCATE (MATRIX(ROWS, COLUMNS)).
    - Since ROWS and COLUMNS are variables, and their values are not known in compiling time,
      - compiler can not generate code like the above one for element reference, how to process this case?
      - Instead, compiler creates a descriptor (called dope vector) for the array. When the storage is allocated for the array, the values of these bounds are calculated and stored in the descriptor. Then the generated code for array reference will use the values from descriptor to calculate relative address.
- Same principle can be used to compile other structured variables. There is a need to store information about the structure of the variables so that correct codes can be generated.

Machine independent compiler features --machine independent code optimization

- Elimination of common subexpressions
  - Code optimization by elimination of common subexpressions (Figure 5.27)
    - 2*J is common subexpression
    - It can be analyzed via quadruples.
    - Quadruple 5 and 12 compute the same value since J does not change. So quadruple 12 can be deleted and change all references to i_{10} to i_3.
    - After i_{10} to i_3, quadruple 6 and 13 become same.
      - So quadruple 13 is removed and substitute i_4 for i_{11}.
    - Similarly, quadruples 10 and 11 can be removed since they are equivalent to 3 and 4.
    - Total number of quadruples: 19→15.
- Code optimization by removal (or move) of loop invariants (Figure 5.27 Cont'd)
  - Even though the total number of quadruples remains same, but the quadruples within the loop body are reduced : 14 → 11.
- Substitution of fast operation for a slower one
  - In the loop, 2**I is substituted by power:=power*2.
  - Quadruples 3,and 4, 3*(I-1) is replaced with disp:=disp+3.
- Other optimizations:
  - Folding: constant computation
  - Loop unrolling: convert loop into line code
  - Loop jamming: merge the bodies of loops.
- Optimizations by programmers:
  - Change **for I=1 to 10 do X[I,2*J-1]:=X[I,2*J]** to:
  - *T1=2*J*;*T2=T1-1*; **for I=1 to 10 do X[I,***T2***]:=X[I,***T1***]**
  - Not good for programmer, also not clear.
  - So let compiler do such kinds of optimizations, have programmers write programs freely and clearly.

Storage allocation

- Variables are assigned storage allocations
  - Programmer defined variables
  - Temporary variables
  - By compiler, so called static allocation.
- Problem with static allocation
  - Recursive procedure call will fail
    - Problem with recursive call of a procedure with static allocation (Figure 5.29)
    - SUB stores the return address for call 3 into RETADR from L, destroying the return address for call 2. Thus, cannot return to MAIN.
    - Same for any variable in SUB. The previous call values are lost.
  - Not allow user to allocate/apply dynamic storage.
    - In C, p=malloc(size); free(p);
- Dynamic allocation (automatic allocation):
  - Activation record: for each procedure call and including: parameters, temporaries, local variables, return address, and register save area.
  - Recursive call will create another activation record.
  - The starting address of an activation record is store in a Base register.
  - Access to the variables by the procedure will be addressed using base relative.
  - Stack is used: when a call is made, its activation record is put on stack. When a call returns, its activation record is deleted from stack. Base register is reset to point to the previous activation record.
  - Recall MASM SS (Stack Segment) register.
  - Compiler will generate code to manage activation records.
    - At the beginning of a procedure, code, called prologue, will create activations on stack
    - At the end of the procedure, code, called epilogue, will delete activation record and resetting pointers.
  - Recursive call of a procedure using dynamic allocation (Figure 5.30) and Figure 5.20 (Cont'd)

Storage allocation (cont.)

- Programmers conducted dynamic allocation:
  - In Pascal: New(p) and dispose(p) and in C: MALLOC(size) and free(p)
  - A variable allocated this way does not have fixed location in memory and must be addressed by indirect addressing via a pointer P. P does have a fix location in activation record.
  - Such kind of storage may be managed by OS or by
  - HEAP, a large space of memory allocated to the user program by OS and managed by a run-time procedure.
  - In some language like Java: there is no need to free the dynamic allocated storage. A run-time garbage collection will do the work.
  - Provides another example of delayed binding: the association of an address with a variable is made when the procedure is executed, not when it is compiled or loaded.
- In general: three kinds of memory:
  - static/global memory for static variables or global variables
  - Stack: for procedure call
  - Heap: for user based dynamic allocation.

Block-structured languages

- A very interesting construct in language
- A block is a portion of a program that has the ability to declare its own identifiers.
  - Nested definitions of a procedure/block within another.
  - A name defined in a block is available within the block including within all the inner blocks defined within this block.
  - A name cannot be used outside its defining block.
  - A same name defined in an inner block will override the one defined in its outer block.
- Nesting of blocks in a source program (Figure 5.31)
  - Ex: in A, VAR X,Y,Z: INTEGER; in B: VAR W,X,Y: REAL; in C: VAR V,W: INTEGER.
  - In B, references to X,Y will be REAL and to Z will be INTEGER
  - In C, references to W will be INTEGER, to X, Y will be REAL, and to Z will be INTEGER.
  - Blocks and their levels, the nesting depth of each block.

Block-structured languages (Cont.)

- Compiling of blocks
  - The entries representing the declaration of the same name declared in different blocks are linked together in symbol table with a chain of pointers.
  - When a reference to a name is encountered, check the current block for its definition, if not, then its directly surrounding block, if not, then the surrounding surrounding block, …
  - Automatic storage allocation: each call to a procedure, its activation record is put on stack.
  - In addition, Display: the display contains pointers to the most recent activation records for the current blocks and all its surrounding blocks in the source program. When a reference to a name declared in a surrounding block, Display is used to find the corresponding activation record in stack.
  - Use of display for procedure in Fig. 5.31 (Figure 5.32)
    - Ex: A→B→C: (a), C→C (b), C→D (c) (since D's direct outside is A), D→B (d) (since B can only access B and A)
  - At the beginning of a block, code is generated to initialize the Display for the block.
  - At the end of a block, code is generated to restore the previous Display.
  - Where is Display?

Compiler Design Options

- One pass compiler
  - Difficult for optimization
  - Declaration of a name appears after its reference.
    - Ex: X:=Y+X, if X, Y, Z are not same type, some conversion code must be generated. Afterward declaration will have problem.
- Multiple passes:
  - ALGOL 68 requires at least three passes.
  - Compilation efficiency vs. execution efficiency
    - E.g., in the student environments, mainly compilation and testing, one pass compilation is good.
    - If execution many times after compilation, or processing large amount of data, multiple passes is OK.
- Interpreters
  - For each statement in source program, translate into some form of intermediate form, and then execute directly. Then to next statement.
  - In general, for statements within a loop, repeatedly analyze and translate the statements.
  - So slower than compilation.
  - Real advantage: debugging. Know the error in source program directly and immediately.
  - So attractive to education environments and beginners.
  - Some cases where interpreters are better:
    - Call to many system libraries, which are the same (already in object code) no matter interpreting or compiling.
    - The types of variables can dynamically change.
    - Dynamic scoping: the variables that can be referred to by a function or a subroutine are determined by the sequence of the calls made during execution rather than the nesting of blocks in the source program.

Compiler Design Options (Cont.)

- P-Code compiler:
  - Intermediate form is the machine language for a hypothetical computer, called pseudo-machine or P-machine.
  - The machine structure is essentially “stack”.
  - Advantage: portability. P-code can be run on any machine with P-code interpreter.
  - Translation and execution using a P-code compiler (Figure 5.33)
    - Main disadvantage: slower.
    - Java: .class.
- Compiler-compiler: compiler generator.
  - Both scanners and parsing can be constructed automatically, in many cases.
  - So, given lexical rules and grammars, along with semantic routines for rules, a compiler can be generated automatically.
  - Automated compiler construction using a compiler-compiler (Figure 5.34)
  - Advantages: reducing the work implementing a compiler,
  - Disadvantages: the compiler generated tends to require more memory and is slower in compilation than the hand-written compiler.
  - However, the code by such compiler may be better since the writer can focus more on optimization.
  - Snow-ball compiler construction technique.

Implementation Examples --SunOS C Compiler

- Pre-processing
  - File inclusion, including assembly subroutines
  - Trigraph sequences conversion, such as ??< to {
  - Line combinations.
  - Lexical analysis
  - Macro processing.
  - Escape sequence conversion, such as \n to a newline
  - Concatenation of adjacent strings. “Hello”, “World” to “Hello, World”.
- Compiled to assembly language program.
- Many system programs are written in C, so efficiency is important.
- Four level code optimizations
  - O1: local optimization; O2: global optimization;
  - O3 and O4: improve speed. Such as loop unrolling (O3), and convert a procedure call into in-line code (O4).
- Insert code in object code to gather running-time information and perform certain tasks.
- As an option, symbolic information can be included in object code (e.g., for debug purpose).

Implementation Examples --Cray MPP FORTRAN Compiler

- Massive Parallel Processing
- Mainly parallelization for shared data.

Implementation Examples --Java Compiler and Environment

- Good for diverse applications such as Internet.
- Object and Object-oriented language.
- Portability.
- No goto and no pointer.
- Automatic garbage collection.
- Built-in support to multiple threads, which are themselves implemented as objects.
- P-code technique:
  - Compile to .class file, which is bytecode for a hypothetical target machine with stack architecture (called Java Virtual Machine, JVM). Java .class can be run/interpreted on any machine supporting JVM, i.e. having a Java interpreter.
  - Java compiler is itself written in Java. Therefore, the compiler can be run on any machine with Java interpreter.
  - A bytecode instruction consists of one byte operation, plus zero or more operands. Many instructions have no explicit operands, instead, they get operands from stack. Stack organization is easy to emulate on any machine with few general-purpose registers.

Implementation Examples --YACC Compiler-Compiler

- Yet Another Compiler-Compiler.
- A parser generator.
  - A BNF grammar specifying syntax and language constructs and set of actions (semantic routines) corresponding to rules in the grammar.
  - Grammars are recognized by pushdown automata.
    - from grammars, generate operator-precedence matrix.
    - Based on matrix, bottom-up technique (called LALR(1)) is used to analyze the token sequences.
    - For < and =, push down to stack,
    - For >, pop up from stack and reduce to a non-terminal (a language construct).
    - Such a pushdown automata, along with its matrix, consists of the parser.
- Associated: LEX, a scanner generator.
  - Given LEX rule: a pattern and its action routine.
  - Patterns are recognized as finite automata.
- Example of input specifications fo LEX and YACC (Figure 5.37)

Summary

- Lexical analysis—scanner
  - String of characters →token
  - Finite automata.
- Syntactical analysis – parser
  - Sequence of tokens –> language constructs
  - Top-down technique – recursive-descent
  - Bottom-up technique– operator-precedence.
  - Grammars, and pushdown automata.
- Intermediate form generation
  - Language constructs → quadruples
  - By semantic routines for language constructs recognized by parser
- Optimizations
  - Use of registers and re-arrangement of instructions
  - Elimination of common subexpressions, removal of loop invariants, and reduction of stronger operations to less ones.
- Object code generations
  - Translated to object code in assembly language or machine instructions.
  - By semantic routines for language constructs recognized by parser
  - Or from optimized intermediate form.
- Other issues:
  - Variables and structured variables.
  - Parameters and temporary variables.
  - Static and dynamic variables.
  - Local and global variables.
  - Static memory, stack, and heap (static, automatic dynamic, user dynamic allocations/features).
  - Call to procedures and recursive call, implemented by stack.
  - Code generated at the beginning of a procedure and the end of the procedure by compiler.
    - Management of activation records and display.
  - Interpreters, P-code, compiler-compiler.
  - The properties of a language is determined/implemented by its compiler.

Summary

--comparison/contrast of assembly and compilation

- Assembling processing vs. compilation
- Assembler vs. compiler
- Assembly language program vs. high language program
  - In assembly language program, the codes and data are interleaved.
  - After compilation, the object program generally consists of codes and data which are separated from codes and are in three different storage allocations: static/global, stack, and heap.
  - Labels (and jmp) and labels (and goto).
  - Why some high languages do not have goto? Why this is possible?