CS 375, Compilers: Class Notes

Gordon S. Novak Jr.

Department of Computer Sciences
University of Texas at Austin
novak@cs.utexas.edu
http://www.cs.utexas.edu/users/novak

[1] A few slides reproduce figures from Aho, Lam, Sethi, and Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley; these have footnote credits.

I wish to preach not the doctrine of ignoble ease, but the doctrine of the strenuous life.

– Theodore Roosevelt

Innovation requires Austin, Texas. We need faster chips and great compilers. Both those things are from Austin.

– Guy Kawasaki

Course Topics

- Introduction
- Lexical Analysis
  - Regular grammars
  - Hand-written lexical analyzer
  - Number conversion
  - Regular expressions
  - LEX
- Syntax Analysis
  - Context-free grammars
  - Operator precedence
  - Recursive descent parsing
  - Shift-reduce parsing, YACC
  - Intermediate code
  - Symbol tables
- Code Generation
  - Code generation from trees
  - Register assignment
  - Array references
  - Subroutine calls
- Optimization
  - Constant folding, partial evaluation
  - Data flow analysis

Pascal Test Program

```
{ program 4.9 from Jensen & Wirth: graph1.pas }

program graph1(output);
const d = 0.0625; {1/16, 16 lines for [x,x+1]}
      s = 32; {32 character widths for [y,y+1]}
      h = 34; {character position of x-axis}
      c = 6.28318; {2*pi}  lim = 32;
var x,y : real;  i,n : integer;
begin
   for i := 0 to lim do
      begin x := d*i; y := exp(-x)*sin(c*x);
      n := round(s*y) + h;
      repeat write(' ');  n := n-1
      until n=0;
      writeln('*')
   end
end.


calling graph1
                                    *
                                          *
                                                *
                                                     *
                                                     *
                                                  *
                                              *
                                         *
                                    *
                               *
                          *
                     *
                     *
                       *
                           *
                               *
                                    *
                                        *
                                           *
                                             *
                                             *
```

Introduction

- What a compiler does; why we need compilers
- Parts of a compiler and what they do
- Data flow between the parts

Machine Language

A computer is basically a very fast pocket calculator attached to a large memory. Machine instructions specify movement of data between the memory and calculator (ALU or Arithmetic/Logic Unit) or tell the ALU to perform operations.

Machine language is the *only* language directly executable on a computer, but it is very hard for humans to write:

- Absolute Addresses: hard to insert code.
- Numeric Codes, e.g. for operations: hard to remember.
- Bit fields, e.g. for registers: hard to pack into numeric form.

Assembly Language

Assembly Language is much easier to program in than Machine Language:

- Addresses are filled in by assembler: makes it easy to insert or remove code.
- Mnemonic codes for operations, e.g. `ADD`.
- Bit fields are handled by assembler.

However, it still is fairly difficult to use:

- One-to-one translation: one output instruction per source line.
  - Programmers write a fixed (small: 8 to 16) number of lines of code per day, independent of language.
  - A programmer costs $2 per minute, $1000 per day!
- Minimal error checking.

High-Level Language

- Higher-level language constructs:
  - Arithmetic Expressions: `x := a + b * c`
  - Control Constructs:
    `while` *expression* `do` *statement*
  - Data Structures:
    `people[i].spouse^.mother`
  - Messages:
    `obj.draw()`
- One-to-many translation: one statement of input generates many machine instructions.
- Cost per machine instruction is much less than using assembly language.
- Error checking, e.g. detection of type errors. Compile-time errors are much cheaper to fix than runtime errors.

Compilers[2]

A compiler translates language X to language Y; "language" is understood very broadly:

- Compile a program to another program. High-level to machine language is the original definition of compiler.
- Compile a specification into a program.
- Compile a graph into a program.
- Translate one realization of an algorithm to another realization.
- Compile a program or specification to hardware.

[2]This slide is by John Werth.

Sequential Phases of a Compiler[3]

Input is a source program.

- Lexical analyzer
- Syntax analyzer
  - Semantic analyzer
  - Intermediate code generator
- Code optimizer
- Code generator

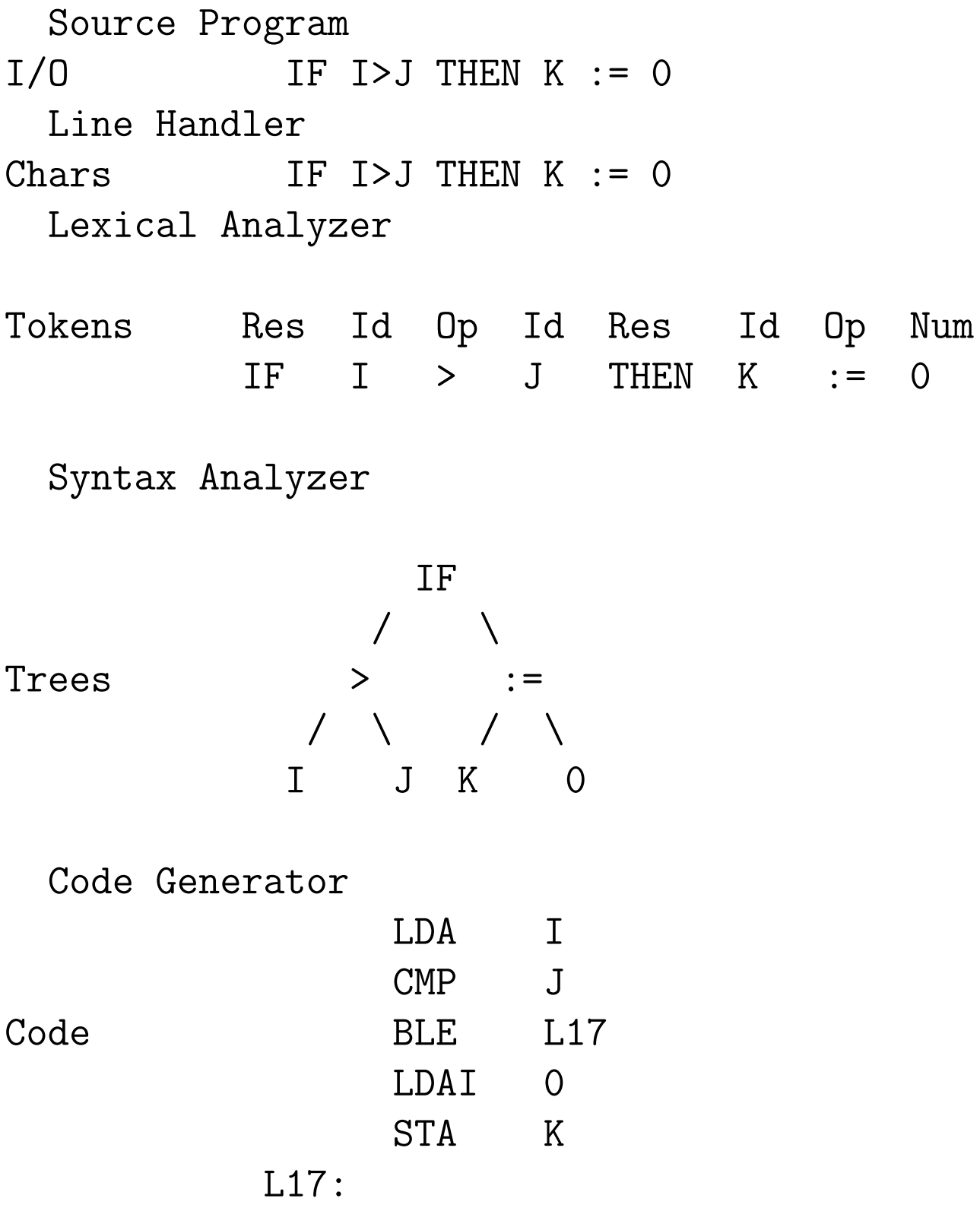
We may think of this as an *analysis* process (understanding what the programmer wants to be done) followed by *synthesis* of a program that performs the intended computation.

These two modules are active throughout the compilation process:

- Symbol table manager
- Error handler

[3]This slide adapted from one by John Werth.

Data Flow through the Compiler



Line Handler

Below the level of the lexical analyzer will be low-level routines that perform input of the source file and get characters from it.

An input line will be treated as an array of characters, with a pointer to the next character (an index in the array).

Interfaces:

- `getchar()` Get the next character from the input line and move the pointer.
- `peekchar()` Get the next character from the input line without moving the pointer.
- `peek2char()` Get the second character from the input line without moving the pointer.

The Line Handler will do such things as skipping whitespace (blanks, tabs, newlines), ignoring comments, handling continuation lines, etc. It may return special "end of statement" or "end of file" pseudo-characters.

Lexical Analyzer

The Lexical Analyzer (or Lexer) will convert characters into "words" or *tokens*, such as:

- Identifiers, e.g. `position`
- Reserved words or keywords, e.g. `begin`
- Numbers, e.g. `3.1415926e2`
- Operators, e.g. `>=`

The Lexical Analyzer may be called as a subroutine such as `gettoken()` to get the next token from the input string. It, in turn, calls the Line Handler routines.

The Lexical Analyzer returns a *token* data structure, consisting of:

- Token Type: identifier, reserved word, number, operator.
- Token Value:
  - Identifiers: string and symbol table pointer
  - Reserved words: integer code.
  - Numbers: internal binary form.
  - Operators: integer code.

Syntactic Analyzer

The Syntactic Analyzer (or Parser) will analyze groups of related tokens ("words") that form larger constructs ("sentences") such as arithmetic expressions and statements:

- `while` *expression* `do` *statement* `;`
- `x := a + b * 7`

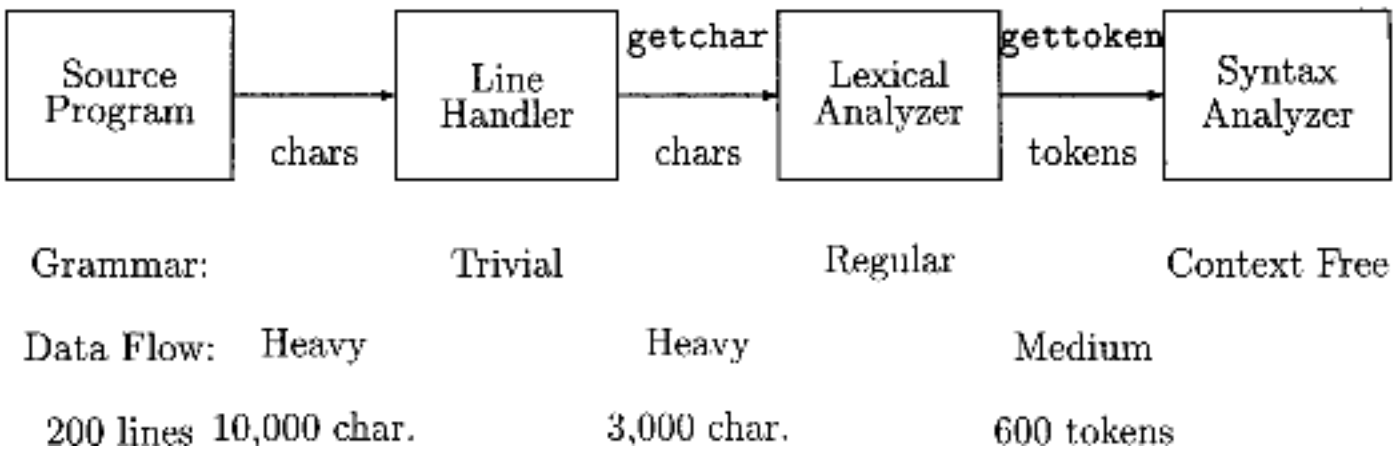
It will convert the *linear* string of tokens into *structured* representations such as expression trees and program flow graphs.

Semantic Analysis

This phase is concerned with the *semantics*, or meaning, of the program. Semantic processing is often performed along with syntactic analysis. It may include:

- Semantic error checking, such as checking for type errors.
- Insertion of extra operations, such as *type coercion* or code for array references.

Lexical Analysis



If speed is needed, the Line Handler and Lexical Analyzer can be coded in assembly language.

The Lexical Analyzer does the following:

- Reads input characters.
- Groups characters into meaningful units or "words", producing data structures called *tokens*.
- Converts units to internal form, *e.g.* converts numbers to machine binary form.
- Serves as a front end for and provides input to the Parser.

Character Classes

At the lowest level of grammar, there is a need to classify characters into classes. This can be done by lookup in an array indexed by the character code. Typical classes include:

- Numerals: `0 1 2 3 4 5 6 7 8 9`
- Alphabetic: `A B C ... Z`
- Whitespace: blank, tab, newline.
- Special: `( ) [ ] + = .` etc.
- Other: characters not in the language `~ @ #`

Special characters may be mapped to consecutive integers to allow the resulting index to be used in `case` statements.

Char	**ASCII**	**Class**
...		
`0`	60_8	`0`
`1`	61_8	`0`
...		
`A`	101_8	`1`
`B`	102_8	`1`
...		

Implementation of Character Classes

Character class names are defined as small-integer constants. A character class array is initialized to map from a character code to the appropriate class.

```
#define ALPHA   1       /* char class names */
#define NUMERIC 2
#define SPECIAL 3

int CHARCLASS[256];     /* char class array */

char specchar[] = "+-*/:=<>^.,;()[]{}";

    for (i = 'a'; i <= 'z'; ++i)   /* init */
       CHARCLASS[i] = ALPHA;
    for (i = '0'; i <= '9'; ++i)
       CHARCLASS[i] = NUMERIC;
    for (i = 0 ; specchar[i] != '\0'; ++i)
       CHARCLASS[specchar[i]] = SPECIAL;
```

The class of a character is looked up in the array:

```
    c = peekchar();
    if (CHARCLASS[c] == ALPHA) ...
```

Hand-written Lexical Analyzer

A lexical analyzer can easily be written by hand. Typically, such a program will call functions `getchar()` and `peekchar()` to get characters from the input.

The lexical analyzer is likewise called as a function, with an entry such as `gettoken()`. The program is structured as:

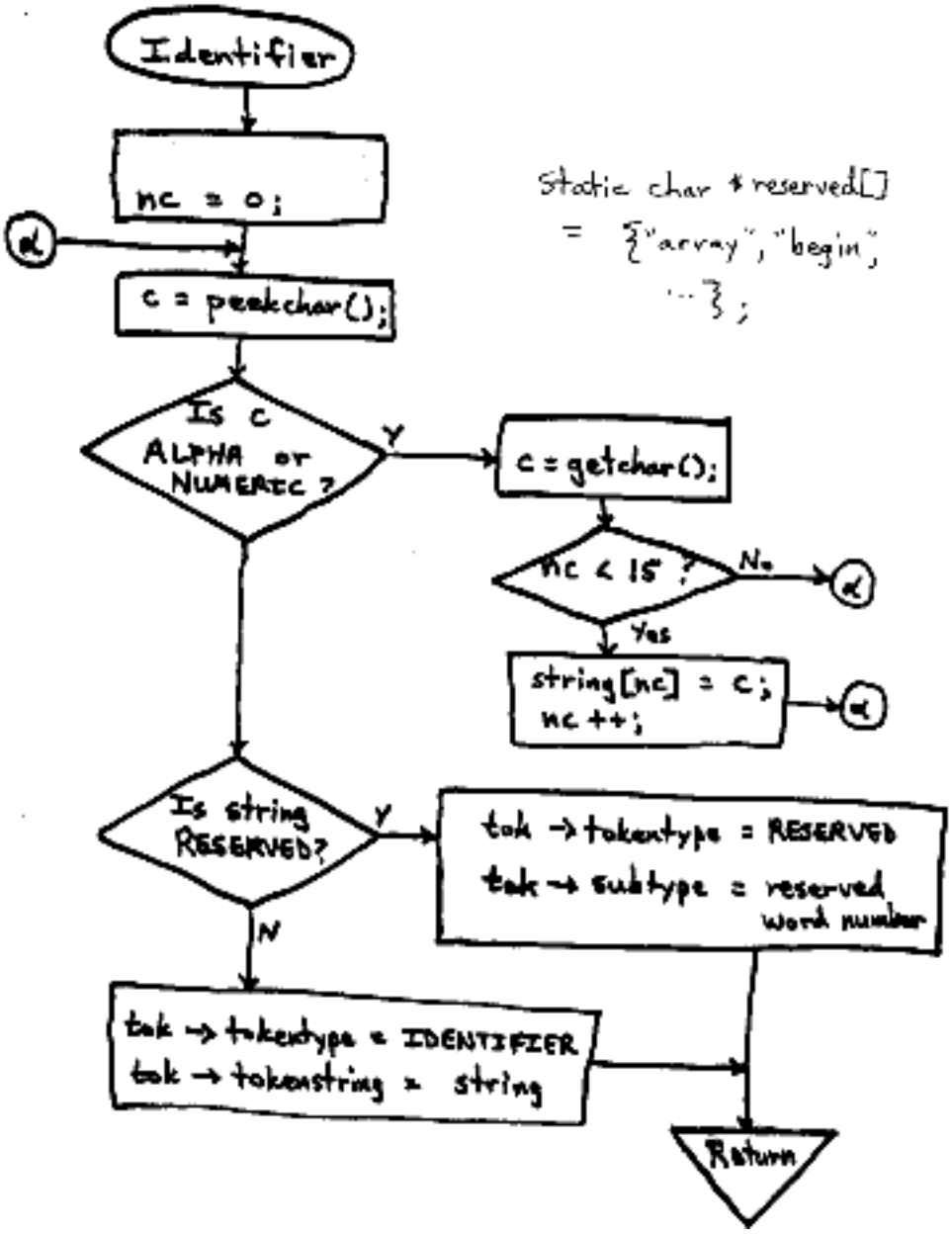
1. A "*big switch*" that skips white space, peeks at the next character to guess what kind of token will be next, and calls the appropriate routine.
2. A set of routines to get particular kinds of tokens, such as identifiers, numbers, strings, or operators.

Typically, a routine will process all tokens that look alike, *e.g.*, all kinds of numbers, or both identifiers and reserved words.

Example Lexical Analyzer

```
/* The ``big switch'': guess token type,
   call a routine to parse it */
TOKEN gettoken()
  {   TOKEN tok;  int c, cclass;
      tok = talloc();   /* allocate new token */
      skipblanks();     /* and comments */
      if ((c = peekchar()) != EOF)
          {
            cclass = CHARCLASS[c];
            if (cclass == ALPHA)
              identifier(tok);
            else if (cclass == NUMERIC)
                    number(tok);
                    else if (c == '\'')
                            getstring(tok);
                            else special(tok);
          }
          else EOFFLG = 1;
     return(tok);
  }
```

Flowchart for Parsing Identifier



Lexical Language Design

A language designer should avoid ambiguity in the design of the lexical syntax of a language.

1. *Reserved words* are a good idea to avoid ambiguity between user symbols and language command words.

   ```
   DO 10 I = 1,25          3.EQ.J
   DO 10 I = 1.25          3.E7+X

   FORMAT(I5)
   FORMAT(I5) = 1.2
   ```

   There should not be too many reserved words.

2. Don't allow spaces inside tokens. Space should never be an operator.

3. Different kinds of tokens should look different at the left end (initial character).

4. Avoid ambiguous combinations of tokens that would require long look-ahead.

5. Avoid "noise" such as ugly special characters. These require extra keystrokes and make programs hard to read. `%rax`

6. Are upper- and lower-case letters equivalent?

Token Data Structure

Converting lexical items into *tokens* simplifies later processing by reducing the input to a few standard kinds of tokens: reserved words, identifiers, numbers, operators, strings, delimiters. The tokens serve as the terminal symbols of the parser grammar.

A token will contain:

1. token type (identifier, operator, etc.)
2. data type: a numeric code indicating `integer`, `real`, etc.
3. pointers to the symbol table
4. pointers for making trees from tokens
5. value of the token (identifier name, number value, numeric code indicating which operator).

Example Token Data Structure

```
typedef struct tokn {
  int   tokentype;  /* OPERATOR, etc */
  int   datatype;   /* INTEGER, REAL, etc */
  struct symtbr * symtype;
  struct symtbr * symentry;
  struct tokn *operands;
  struct tokn *link;
  union { char  tokenstring[16];
          int   which;
          long  intnum;
          float realnum; } tokenval;
  } TOKENREC, *TOKEN;
```

`symtype`	pointer to type in symbol table
`symentry`	pointer to variable in symbol table
`operands`	down pointer to operand tokens
`link`	side pointer to sibling token
`whichval`	integer code: which operator, etc.
	`define`d as `tokenval.which`
`stringval`	string constant or variable name
`intval`	value of integer constant
`realval`	value of real constant

Number Conversion

Arabic (Indian) numerals are written in the form $a_n a_{n-1} ... a_1 a_0$ denoting, in number base r, the integer:

$a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + ... + a_1 \cdot r^1 + a_0$

Factoring this expression yields:

$((...((0 \cdot r + a_n) \cdot r + a_{n-1}) \cdot r + ...) \cdot r + a_1) \cdot r + a_0$

This suggests an algorithm for converting a number expressed by digits $d_n d_{n-1} ... d_1 d_0$ to internal form in a left-to-right scan:

1. Initialize the accumulator, `num = 0`.
2. For each new digit, d_i, let `a` be the number denoted by d_i:
   In C, (`d_i - '0'`)
   In Lisp, `(- (char-code` d_i`) (char-code #\0))`
   Then set `num = num * r + a` .
3. After all digits have been processed, `num` is the numeric value in internal form.

Simple Number Scanner

```
void number (TOKEN tok)
  { long num;
    int  c, charval;
    num = 0;
    while ( (c = peekchar()) != EOF
            && CHARCLASS[c] == NUMERIC)
      {   c = getchar();
          charval = (c - '0');
          num = num * 10 + charval;
        }
    tok->tokentype = NUMBERTOK;
    tok->datatype = INTEGER;
    tok->intval = num;
  }
```

Lexical Analyzer Output

```
Started scanner test.
tokentype:  2  which:   19      program
tokentype:  3  value:          graph1
tokentype:  1  which:    4            (
tokentype:  3  value:          output
tokentype:  1  which:    5            )
tokentype:  1  which:    2            ;
tokentype:  2  which:    4        const
tokentype:  3  value:                 d
tokentype:  0  which:    6            =
tokentype:  5  type:     1 6.250000e-02
tokentype:  1  which:    2            ;
tokentype:  3  value:                 s
tokentype:  0  which:    6            =
tokentype:  5  type:     0           32
tokentype:  1  which:    2            ;
tokentype:  3  value:                 h
tokentype:  0  which:    6            =
tokentype:  5  type:     0           34
tokentype:  1  which:    2            ;
tokentype:  3  value:                 c
tokentype:  0  which:    6            =
tokentype:  5  type:     1 6.283180e+00
tokentype:  1  which:    2            ;
```

Floating Point Numbers

Numbers containing a decimal point can be converted in a manner similar to that used for integers.

The important thing to note is that the decimal point is only a place marker to denote the boundary between the integer and fraction parts.

1. Convert the number to an integer as if the decimal point were not present.
2. Count the number of digits after the decimal point has been found.
3. Include only an appropriate number of significant digits in the mantissa accumulation.
4. Leading zeros are not significant, but must be counted if they follow the decimal point.
5. At the end:
   (a) Float the accumulated mantissa
   (b) Combine the digit counts and the specified exponent, if any.
   (c) Multiply or divide the number by the appropriate power of 10 (from a table).

IEEE Floating Point Standard

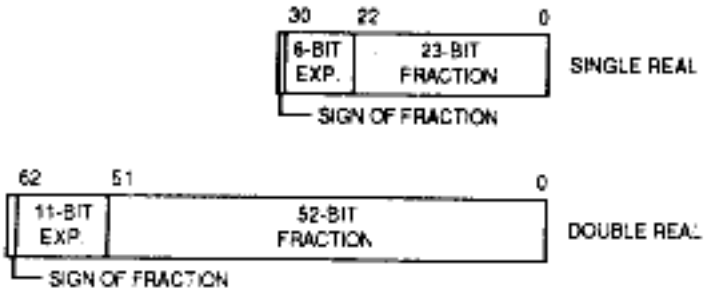


Figure 2-9. Memory Formats for Real Data Types

The exponent in all three binary formats is an unsigned binary integer with an implied bias added to it. The bias values for single, double, and extended precision are 127, 1023, and 16383, respectively. When the bias is subtracted from the value of the exponent, the result represents a signed, twos complement power of two which, when multiplied by the mantissa, yields the magnitude of a normalized floating-point number. Note that the use of biased exponents allows floating-point numbers in memory to be compared using the M68000 Family integer compare instruction (CMP), regardless of the absolute magnitude of the exponents.

Data formats for single and double precision numbers differ slightly from the data format for extended precision numbers in the representation of the mantissa. A normalized mantissa, for all three precisions, is always in the range [1.0...2.0). The extended precision data format explicitly represents the entire mantissa, including the explicit integer part bit. However, for single and double precision data formats, only the fractional portion of the mantissa is explicitly represented and the integer part is always one. Thus, the integer part bit is implicit for single and double precision formats.

The IEEE standard has created the term "significand" to bridge this difference and to avoid the historical implications of the term mantissa. The IEEE standard defines a significand as that component of a binary floating-point number which consists of an explicit or implicit leading bit to the left of the implied binary point and a fraction field to the right of the implied binary point.

This manual uses the terms mantissa and significand interchangeably, given the relationships as shown below.

Single Precision Mantissa	=	Single Precision Significand
	=	1.<23-Bit Fraction Field>
Double Precision Mantissa	=	Double Precision Significand
	=	1.<52-Bit Fraction Field>

Floating Point Examples

```
/*  floats.exmp    Print out floating point numbers         06 Feb 91   */
static float nums[30] = { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0,
 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 3.1, 3.14,
 3.1415927, 0.5, 0.25, 0.125, -1.0, -2.0, -3.0, -0.5, -0.25, -3.1415927 };

printnum(f,plainf)
  float f;  unsigned plainf;    /* look at the float as a bit string */
  {  int sign, exp, expb;  long mant, mantb;
  { sign = (plainf >> 31) & 1;
    exp  = (plainf >> 20) & 2047;
    expb = exp - 1023;
    mant = plainf & 1048575;
    mantb = mant + 1048576;
    printf("%12f  %11o    %1o    %4o       %5d     %7o       %7o\n",
           f, plainf, sign, exp, expb, mant, mantb);  } }

/* This appears to be double-precision floating point format: */
/* 1 bit sign, 11 bits biased exponent, 20 bits mantissa + 32 in next word */
```

floating	octal	sign	biased exponent	corrected exponent	actual mantissa	corrected mantissa
0.000000	0	0	0	-1023	0	4000000
1.000000	7774000000	0	1777	0	0	4000000
2.000000	10000000000	0	2000	1	0	4000000
3.000000	10002000000	0	2000	1	2000000	6000000
4.000000	10004000000	0	2001	2	0	4000000
5.000000	10005000000	0	2001	2	1000000	5000000
9.000000	10010400000	0	2002	3	400000	4400000
3.100000	10002146314	0	2000	1	2146314	6146314
3.140000	10002217270	0	2000	1	2217270	6217270
3.141593	10002220773	0	2000	1	2220773	6220773
0.500000	7770000000	0	1776	-1	0	4000000
0.250000	7764000000	0	1775	-2	0	4000000
0.125000	7760000000	0	1774	-3	0	4000000
-1.000000	27774000000	1	1777	0	0	4000000
-2.000000	30000000000	1	2000	1	0	4000000
-3.000000	30002000000	1	2000	1	2000000	6000000
-0.500000	27770000000	1	1776	-1	0	4000000
-3.141593	30002220773	1	2000	1	2220773	6220773

Errors[4]

Several kinds of errors are possible:

- Lexical: `x := y ~ z`
  The character `~` is not allowed in Pascal.
- Syntactic: `x := y  z`
  There is no operator between `y` and `z`.
- Semantic: `x := y mod 3.14`
  The operator `mod` requires `integer` arguments.

The seriousness of errors can vary:

- Diagnostic: not necessarily an error, but might be: `x == 3.14` may not be a meaningful comparison.
- Error: definitely an error; code generation will be aborted, but compilation may continue.
- Fatal error: so bad that the compiler must stop immediately.

Cascading errors occur when one real error causes many reported errors, e.g. forgetting to declare a variable can cause an error at each use.

[4]This slide adapted from one by John Werth.

Error Messages

The compiler writer has a serious obligation: the compiler *must* produce either correct output code or an error message.

Good error messages can save a great deal of programmer time; this makes it worth the trouble to produce them.

1. The message should be written out as text.
2. A pointer to the point of the error in the input program should be provided when appropriate.
3. Values from the program should be included in the message where appropriate.
4. Diagnostic messages (*e.g.*, unused variables) should be included, but user should be able to turn them off.

```
X[CVAR] := 3.14

   ↑

*ERROR*  CVAR, of type COMPLEX,
         may not be used as a subscript.
```

Formal Syntax

There is a great deal of mathematical theory concerning the syntax of languages. This theory is based on the work of Chomsky.

Formal syntax is better at describing artificial languages such as programming languages than at describing natural languages.

Grammar

A *grammar* specifies the legal syntax of a language. The kind of grammar most commonly used in computer language processing is a *context-free grammar*. A grammar specifies a set of *productions*; *non-terminal symbols* (phrase names or parts of speech) are enclosed in angle brackets. Each production specifies how a nonterminal symbol may be replaced by a string of terminal or nonterminal symbols, e.g., a Sentence is composed of a Noun Phrase followed by a Verb Phrase.

```
<S>      -->  <NP> <VP>
<NP>     -->  <ART> <ADJ> <NOUN>
<NP>     -->  <ART> <NOUN>
<NP>     -->  <ART> <NOUN> <PP>
<VP>     -->  <VERB> <NP>
<VP>     -->  <VERB> <NP> <PP>
<PP>     -->  <PREP> <NP>

<ART>    -->   A | AN | THE
<NOUN>   -->   BOY | DOG | LEG | PORCH
<ADJ>    -->   BIG
<VERB>   -->   BIT
<PREP>   -->   ON
```

Language Generation

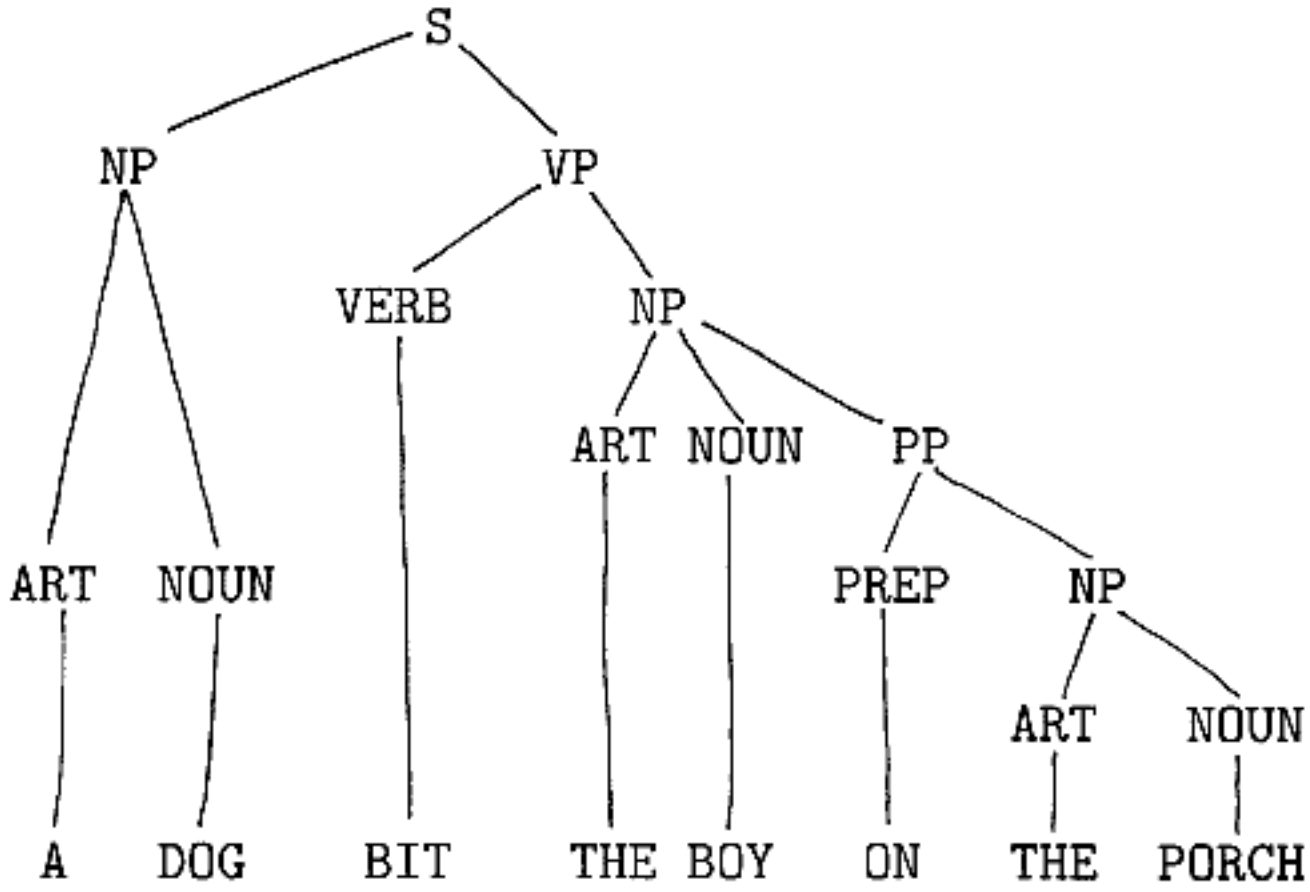
Sentences can be generated from a grammar by the following procedure:

- Start with the sentence symbol, <S>.
- Repeat until no nonterminal symbols remain:
  - Choose a nonterminal symbol in the current string.
  - Choose a production that begins with that nonterminal.
  - Replace the nonterminal by the right-hand side of the production.

```
<S>
<NP> <VP>
<ART> <NOUN> <VP>
THE <NOUN> <VP>
THE DOG <VP>
THE DOG <VERB> <NP>
THE DOG <VERB> <ART> <NOUN>
THE DOG <VERB> THE <NOUN>
THE DOG BIT THE <NOUN>
THE DOG BIT THE BOY
```

Parsing

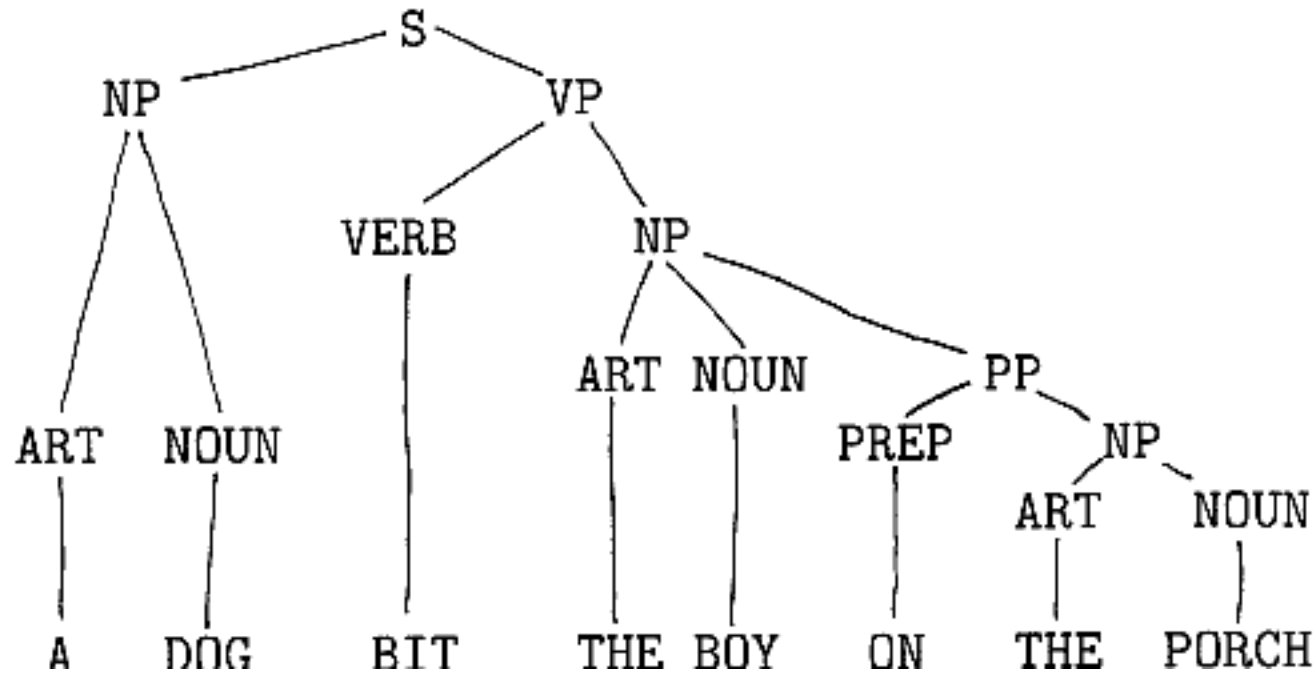
Parsing is the inverse of generation: the *assignment of structure* to a linear string of words according to a grammar; this is much like the "diagramming" of a sentence taught in grammar school.

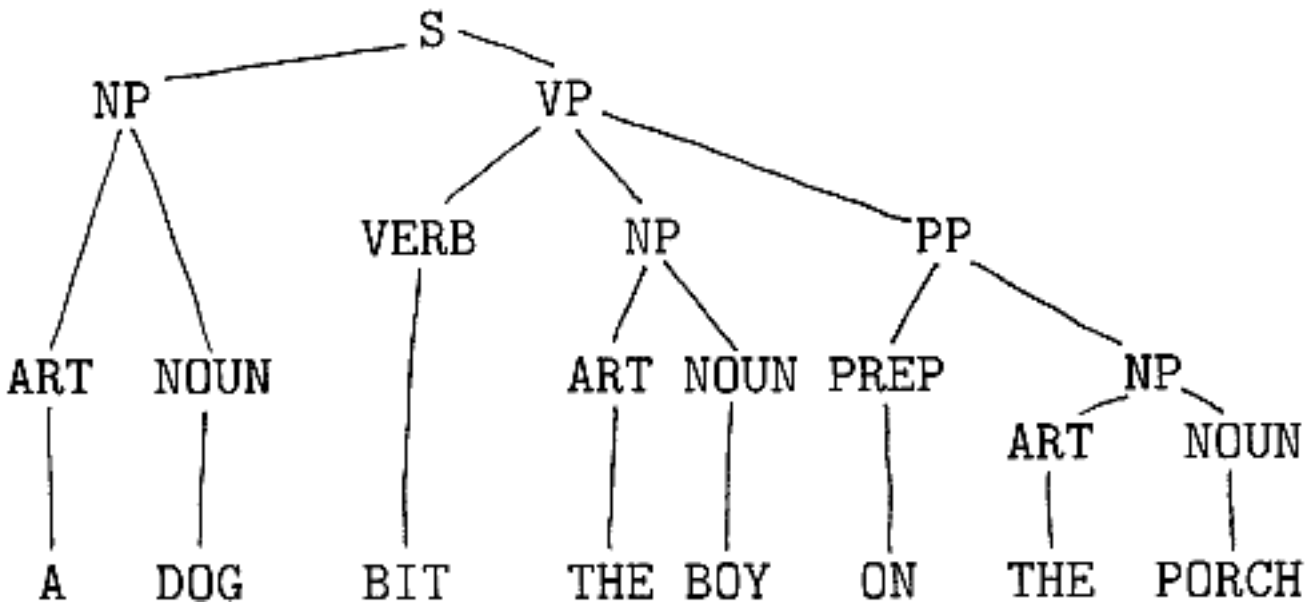


Parts of the *parse tree* can then be related to object symbols in the computer's memory.

Ambiguity

Unfortunately, there may be many ways to assign structure to a sentence (e.g., what does a PP modify?):





Notation

The following notations are used in describing grammars and languages:

V^*	Kleene closure: a string of 0 or more elements from the set V.
V^+	1 or more elements from V
$V^?$	0 or 1 elements from V (*i.e.*, optional)
$a\|b$	either a or b
$< nt >$	a nonterminal symbol or phrase name
ϵ	the empty string

Phrase Structure Grammar

A grammar describes the structure of the sentences of a language in terms of components, or phrases. The mathematical description of phrase structure grammars is due to Chomsky.[5]

Formally, a *Grammar* is a four-tuple $G = (T, N, S, P)$ where:

- **T** is the set of *terminal symbols* or *words* of the language.
- **N** is a set of *nonterminal symbols* or *phrase names* that are used in specifying the grammar. We say $V = T \cup N$ is the *vocabulary* of the grammar.
- **S** is a distinguished element of N called the *start symbol*.
- **P** is a set of *productions*, $P \subseteq V^*NV^* \times V^*$. We write productions in the form $a \rightarrow b$ where a is a string of symbols from V containing at least one nonterminal and b is any string of symbols from V.

[5]See, for example, Aho, A. V. and Ullman, J. D., *The Theory of Parsing, Translation, and Compiling*, Prentice-Hall, 1972; Hopcroft, J. E. and Ullman, J. D., *Formal Languages and their Relation to Automata*, Addison-Wesley, 1969.

Chomsky Hierarchy

Chomsky defined 4 classes of languages, each of which is a proper superset of the rest:

Type 0: General Phrase-structure
Type 1: Context Sensitive
Type 2: Context Free
Type 3: Regular

These languages can be characterized in several ways:

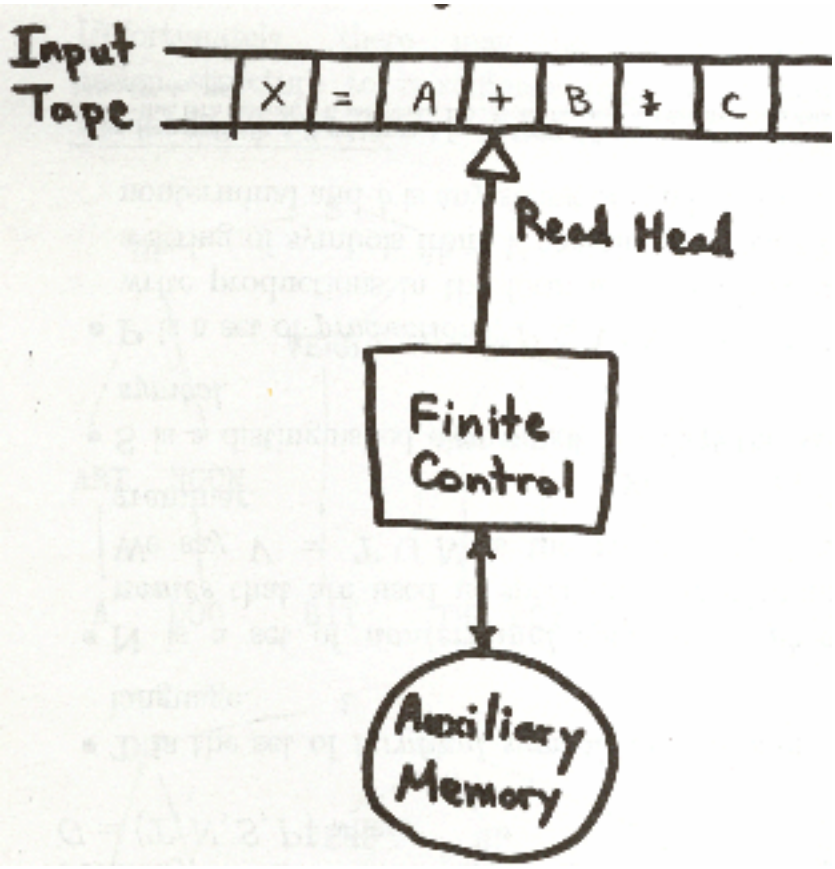
- Type of allowable productions in the grammar
- Type of recognizing automaton
- Memory required for recognition

Recognizing Automaton

A *recognizing automaton* is an abstract computer that reads symbols from an input tape. It has a finite control (computer program) and an auxiliary memory.

The recognizer answers "Yes" or "No" to the question "Is the input string a member of the language?"

The kinds of languages that can be recognized depend on the amount of auxiliary memory the automaton has (finite, pushdown stack, tape whose size is a linear multiple of input length, infinite tape).



Chomsky Language Hierarchy

Type	Name	Allowable Productions	Example Language	Example Grammar
0	Type 0	Unrestricted		
1	Context Sensitive	$\alpha \to \beta$ where $\lvert\alpha\rvert \leq \lvert\beta\rvert$ $\alpha \in V^*NV^*$ $\beta \in V^+$	$a^n b^n c^n$	$S \to aSBC$ $S \to aBC$ $CB \to BC$ $aB \to ab$ $bB \to bb$ $bC \to bc$ $cC \to cc$
2	Context Free	$A \to \alpha$ $A \in N$ $\alpha \in V^*$	$a^n b^n$	$S \to aSb$ $S \to ab$
3	Regular Right Linear Finite Automaton Recognizable	$A \to xB$ $A \to x$ $A, B \in N$ $x \in T^*$	$a^n b$	$S \to ab$ $S \to aS$

Regular Languages

Productions: $A \to xB$
$A \to x$
$A, B \in N$
$x \in T^*$

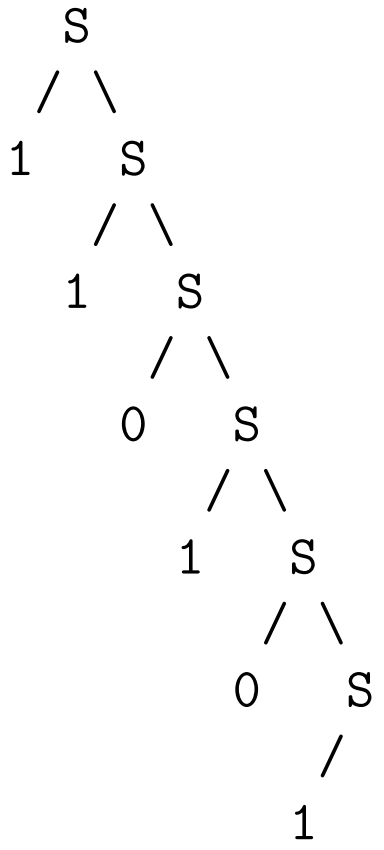
- Only *one* nonterminal can appear in any derived string, and it must appear at the right end.
- Equivalent to a *deterministic finite automaton* (simple program).
- Parser never has to back up or do search.
- Linear parsing time.
- Used for simplest items (identifiers, numbers, word forms).
- Any *finite* language is regular.
- Any language that can be recognized using finite memory is regular.

Example Regular Language

A binary integer can be specified by a regular grammar:

$$
\begin{aligned}
&< S > \ \rightarrow \ \texttt{0} < S > \\
&< S > \ \rightarrow \ \texttt{1} < S > \\
&< S > \ \rightarrow \ \texttt{0} \\
&< S > \ \rightarrow \ \texttt{1}
\end{aligned}
$$

The following is a parse tree for the string `110101`. Note that the tree is linear in form; this is the case for any regular language.



lex

`lex` is a lexical analyzer generator, part of a compiler-compiler system when paired with `yacc`.[6]

`lex` allows a lexical analyzer to be constructed more easily than writing one by hand. `lex` allows the grammar to be specified using *regular expressions*; these are converted to a nondeterministic finite automaton (NFA), which is converted to a deterministic finite automaton (DFA), which is converted to tables to control a table-driven parser.

`lex` reads a source file, named using a `.l` suffix, compiles it, and produces an output file that is always called `lex.yy.c` . This is a C file that can be compiled using the C compiler to produce an executable.

[6]There are Gnu versions of `lex` and `yacc` called `flex` and `bison`. These are mostly, but not completely, compatible with `lex` and `yacc`.

Regular Expressions

Regular expressions are a more convenient way (than a regular grammar) to specify a regular language. We will use `lex` conventions for specifying regular expressions. An expression is specified in left-to-right order.

Expression:	**Meaning:**
`[` *chars* `]`	Any member of the set of characters *chars*.
`[` c_1 `-` c_2 `]`	Any character from c_1 through c_2.
`[^` *chars* `]`	Any character except *chars*.
`(` *specs* `)`	Used to group specifications *specs*.
`{` *category* `}`	An instance of a previously named *category*.
`"` *string* `"`	Exactly the specified *string*.
s_1 `|` s_2	s_1 or s_2
spec `*`	Zero or more repetitions of *spec*.
spec `+`	One or more repetitions of *spec*.
spec `?`	Optional *spec*.
spec `{` m, n`}`	m through n repetitions of *spec*.

Lex Specifications[7]

```
%{  declarations  %}
regular definitions
%%
translation rules
%%
auxiliary procedures
```

- declarations: `include` and manifest constants (identifier declared to represent a constant).
- regular definitions: definition of named syntactic constructs such as `letter` using regular expressions.
- translation rules: pattern / action pairs
- auxiliary procedures: arbitrary C functions copied directly into the generated lexical analyzer.

[7] slide by John Werth.

Sample lex Specification[8]

```
%{ /* lexasu.l  Fig. 3.23 from Aho, Lam, Sethi, and Ullman, Compilers */

#define LT 8      /* Example of use:                                    */
#define LE 9      /* lex /projects/cs375/lexasu.l  compile lexasu.l to C   */
#define EQ 6      /* cc lex.yy.c -ll             Compile lex output with C */
#define NE 7      /* a.out                       Execute C output          */
#define GT 11     /* if switch then 3.14 else 4  Test data                 */
#define GE 10     /* ^D                          Control-D for EOF to stop */
#define ID      3
#define NUMBER  5
#define OP      1 /* to avoid returning 0 */
#define IF     13
#define THEN   23
#define ELSE    7
int yylval;       /* type of the returned value */

%}   /* regular definitions */

delim   [ \t\n]
ws      {delim}+
letter  [A-Za-z]
digit   [0-9]
id      {letter}({letter}|{digit})*
number  {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%

{ws}     { /* no action and no return */ }
if       { return(IF); }
then     { return(THEN); }
else     { return(ELSE); }
{id}     { yylval = install_id(); return(ID); }
{number} { yylval = install_num(); return(NUMBER); }
"<"      { yylval = LT; return(OP); }
"<="     { yylval = LE; return(OP); }
"="      { yylval = EQ; return(OP); }
"<>"     { yylval = NE; return(OP); }
">"      { yylval = GT; return(OP); }
">="     { yylval = GE; return(OP); }
```

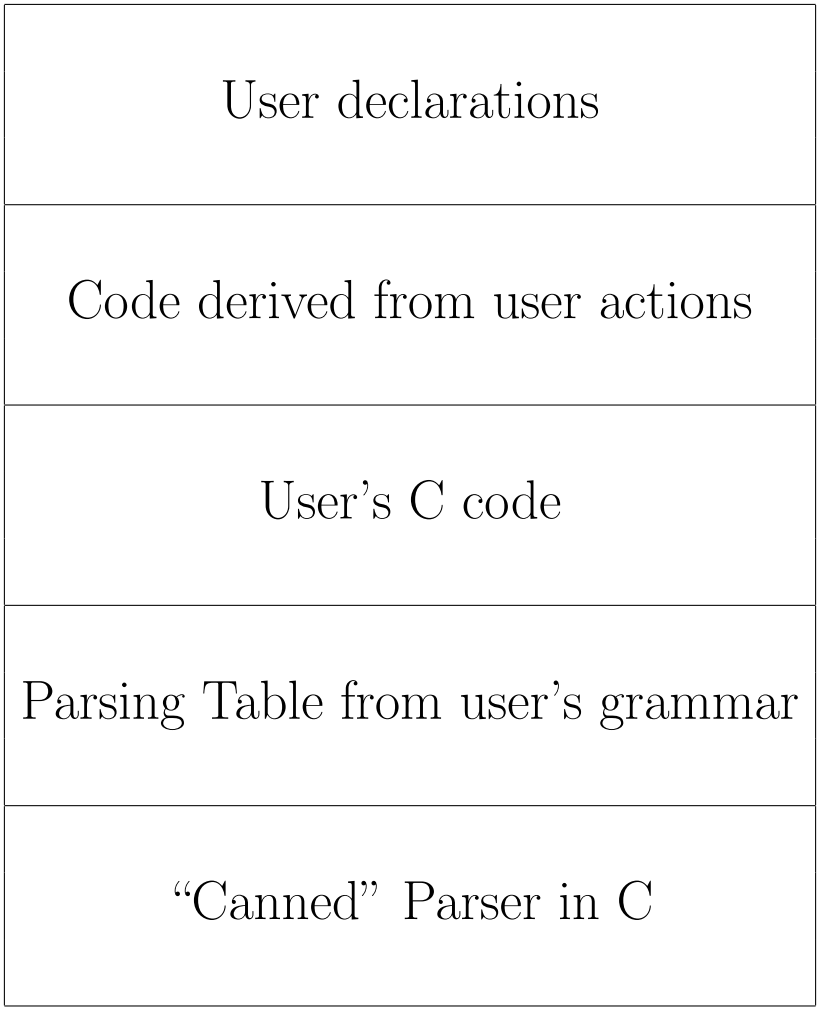
[8]Runnable version of Fig. 3.23 from Aho, Lam, Sethi, and Ullman, *Compilers*.

C for Lex Sample

```
%%       /* C functions */

install_id() {  printf("id  %10s   n = %4d\n",yytext,yyleng);  }

install_num() {  printf("num %10s   n = %4d\n",yytext,yyleng);  }

yywrap() {  return(1);  }   /* lex seems to need this. */

void main()          /* Call yylex repeatedly to test */
  { int res, done;
    done = 0;
    while (done == 0)
      { res = yylex();
        if (res != 0)
           {
            printf("yylex result = %4d\n", res);
           }
           else done = 1;
      }
    exit(0);
    }
```

lex.yy.c

The file `lex.yy.c` produced by `lex` has the following structure (different versions of `lex` may put the sections in different orders).



Comments on Sample lex[9]

Manifest Constants: these definitions are surrounded by `%{ ... %}` and will be copied verbatim into the generated program.

Regular Definitions: these are names followed by a regular expression. For example, `delim` is one of the characters blank, tab, or newline.

Note that if a string is a name then it is surrounded by braces (as `delim` is in the definition of `ws`) so that it will not be interpreted as a set of characters.

`[A-Z]` is the set of characters from A to Z.

Parentheses are meta-symbols used to group. `|` is a meta-symbol for union. `?` is a meta-symbol for 0 or more occurrences. `-` is a meta-symbol for range.

`\` is an escape which allows a meta-symbol to be used as a normal character. `""` has the same effect.

[9] slide by John Werth.

Translation Section[10]

The `{ws}` rule causes the lexical analyzer to skip all delimiters until the next non-delimiter.

The `if` rule recognizes the string 'if'. When it is found the lexical analyzer returns the *token* `IF`, a manifest constant.

The `{id}` rule must do three jobs:

1. record the id in the symbol table
2. return a pointer to the specific id
3. return the token `ID` to signify that an id was seen

The first two are accomplished by

```
yylval = install(id);
```

`yylval` is a global used for this purpose in Yacc. The third is accomplished by `return(ID);`

The action for `{number}` is similar.

The rules for the relational operators set `yylval` to specific manifest constants to show which value was found and return the token `RELOP` to signify that a relational operator was seen.

[10] slide by John Werth.

Lex Conventions[11]

- The program generated by Lex matches the longest possible prefix of the input. For example, if `<=` appears in the input then rather than matching only the `<` (which is also a legal pattern) the entire string is matched.
- Lex keywords are:
  - `yylval`: value returned by the lexical analyzer (pointer to token)
  - `yytext`: pointer to lexeme (array of characters that matched the pattern)
  - `yyleng`: length of the lexeme (number of chars in `yytext`).
- If two rules match a prefix of the same and greatest length then the first rule appearing (sequentially) in the translation section takes precedence.

  For example, `if` is matched by both `if` and `{id}`. Since the `if` rule comes first, that is the match that is used.

[11] slide by John Werth.

The Lookahead Operator[12]

If `r1` and `r2` are patterns, then `r1/r2` means match `r1` only if it is followed by `r2`.

For example,

```
DO/({letter}|{digit})*=({letter}|{digit})*,
```

recognizes the keyword `DO` in the string `DO5I=1,25`

[12] slide by John Werth.

Auxiliary Procedures[13]

In this third section the user may insert any desired code in the generated program.

This typically will include symbol table routines of some kind. These routines will use/work with the lex-defined globals `yylval`, `yytext`, and `yyleng`.

[13]slide by John Werth.

Parser Overview

The *Parser* is a central part of a compiler.

- The input to the parser is the output of the lexical analyzer (`gettoken`).
- The parser analyzes whole statements of the program: `if` *expression* `then` *statement* `else` *statement*
- Since the language constructs may be recursive, a context-free grammar must be used.
- The parser builds complex tables, such as the symbol table, in response to declaration statements. These tables are used later in the generation of code.
- The output of the parser is *intermediate code*.

Context Free Languages

Productions: $A \to \alpha$
$A \in N$
$\alpha \in V^*$

- Since left-hand-side of each production is a single nonterminal, every derivation is a tree.
- Many good parsers are known. Parsing requires a recursive program, or equivalently, a stack for temporary storage.
- Parsing time is at worst $O(n^3)$, though programming languages are commonly $O(n)$.
- Used for language elements that can contain themselves, e.g.,
  - Arithmetic expressions can contain subexpressions: $A + B * (C + D)$.
  - A noun phrase can contain a prepositional phrase, which contains a noun phrase:
    a girl with a hat on her head.
- Any language that requires counting must be at least context free: $a^n b^n$, balanced parentheses.

Context Sensitive Languages

Productions: $\alpha \rightarrow \beta$

$\alpha \in V^* N V^*$

$\beta \in V^+$

$|\alpha| \leq |\beta|$

The strings around the N on the left-hand side of the production are the *context*, so a production works only in a particular context and is therefore context sensitive.

- Context sensitivity seems applicable for some aspects of natural language, e.g., subject-verb agreement.

  John likes Mary.

  $*$ John like Mary.

- No effective parsing algorithm is known.
- Parsing is NP-complete, i.e., may take exponential time, requires a search.
- Context sensitive languages are not used much in practice.

Derivations

A *derivation* is the process of deriving a sentence from the start symbol S according to a grammar, *i.e.*, the replacement of nonterminal symbols by the right-hand sides of productions.

We use the symbol $\Rightarrow$ to denote a derivation step.

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

shows the steps in deriving the expression $-(id)$ using an expression grammar.

$\Rightarrow^*$ is used to denote derivation in zero or more steps, while $\Rightarrow^+$ is used to denote derivation in one or more steps.

In a *leftmost derivation*, the leftmost nonterminal is replaced in each step. If $S \Rightarrow_{lm}^* \alpha$, then α is called a *left-sentential form* of the grammar G. A leftmost derivation is traced by a predictive, top-down parser.

In a *rightmost* (or *canonical*) derivation, the rightmost nonterminal is replaced in each step. A shift-reduce parser (*e.g.* YACC) traces a rightmost derivation "backwards".

Language Generated by a Grammar

Given a grammar G with start symbol S, the *language generated by* G, denoted $L(G)$, is the set of derivable terminal strings w:

$$L(G) = \{w \mid S \stackrel{+}{\Rightarrow} w\}$$

Two grammars that generate the same language are *equivalent*.

$$a^*b^*(a|b)^* = (a|b)^*$$

The *union* of two languages is simply set union, *i.e.*,

$$L_1 \cup L_2 \;=\; \{l \mid l \in L_1 \vee l \in L_2\}$$

The *concatenation* of two languages is:

$$L_1 L_2 \;=\; \{l_1 l_2 \mid l_1 \in L_1 \wedge l_2 \in L_2\}$$

Ambiguity and Left Recursion

Many derivations could correspond to a single parse tree. A grammar for which some sentence has more than one parse tree is *ambiguous*. An ambiguous grammar can be dealt with in two ways:

- Rewrite the grammar to be one that generates the same language but is unambiguous. For example, the `if-then-else` ambiguity can be eliminated.
- Use disambiguating rules to guide the parser. LEX and YACC have defaults that handle many common cases correctly. Operator precedence parsing is a form of disambiguating rules.

A grammar is *left recursive* iff $A \stackrel{+}{\Rightarrow} A\alpha$ for some nonterminal A. A left recursive grammar can cause a top-down recursive descent parser to go into an infinite loop. Left recursion can be eliminated by *left factoring* to obtain an equivalent grammar.

Parsing

A *parser* is a program that converts a *linear* string of input words into a *structured* representation that shows how the phrases (substructures) are related and shows how the input could have been derived according to the grammar of the language.

Finding the correct parsing of a sentence is an essential step towards extracting its meaning.

There are several different kinds of parsers:

- Top-down
- Bottom-up
- Chart
- Augmented transition network

Top-down Parser

A *top-down* parser begins with the Sentence symbol, `S`, expands a production for `S`, and so on recursively until words (terminal symbols) are reached. If the string of words matches the input, a parsing has been found.[14]

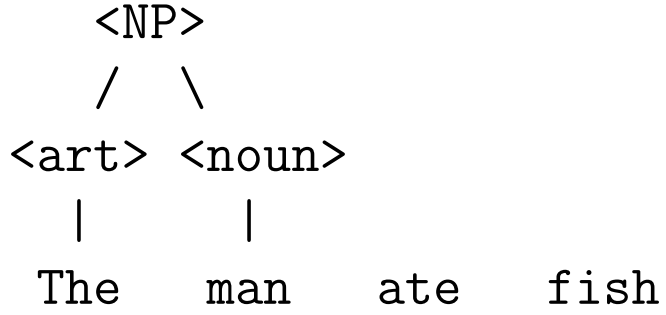
This approach to parsing might seem hopelessly inefficient. However, *top-down filtering*, that is, testing whether the next word in the input string could begin the phrase about to be tried, can prune many failing paths early.

For languages with *keywords*, such as programming languages or natural language applications, top-down parsing can work well. It is easy to program.

[14] See the Language Generation slide earlier in this section.

Bottom-up Parsing

In *bottom-up* parsing, words from the input string are reduced to phrases using grammar productions:



This process continues until a group of phrases can be reduced to `S`.

Chart Parser

A *chart parser* is a type of bottom-up parser that produces all parses in a triangular array called the *chart*; each chart cell contains a set of nonterminals. The bottom level of the array contains all possible parts of speech for each input word. Successive levels contain reductions that span the items in levels below: cell $a_{i,k}$ contains nonterminal N iff there is a parse of N beginning at word i and spanning k words.

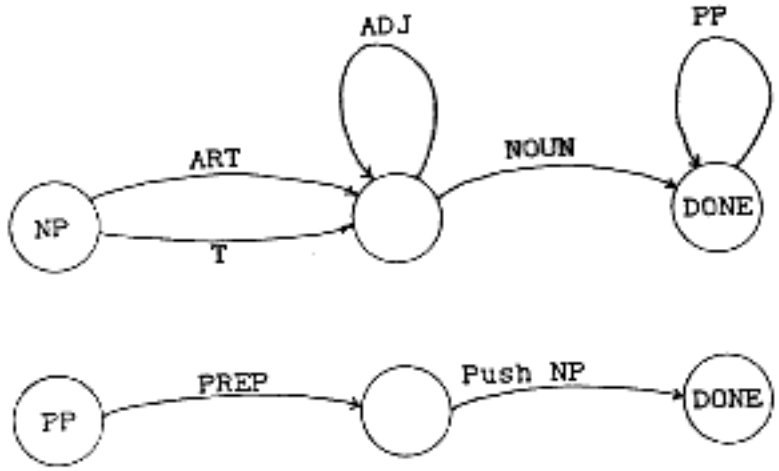
5	S				
4					
3	NP		VP		
2	NP			NP	
1	Art	Adj, N	N, V	Art	N
0	The	old	man	the	boats
	1	2	3	4	5

The chart parser eliminates the redundant work that would be required to reparse the same phrase for different higher-level grammar rules.

The Cocke-Kasami-Younger (CKY) parser is a chart parser that guarantees to parse any context-free language in at most $O(n^3)$ time.

Augmented Transition Network Grammars

A grammar can be written in network form. Branches are labeled with parts of speech or phrase names. Actions, such as constructing a database query, can be taken as arcs are traversed.



ATN's are more readable than lists of productions; they can easily be coded in Lisp.

Augmented Transition Networks

An ATN [15] is like a finite state transition network, but is augmented in three ways:

1. **Arbitrary tests** can be added to the arcs. A test must be satisfied for the arc to be traversed. This allows, for example, tests on agreement of a word and its modifier.
2. **Structure-building actions** can be added to the arcs. These actions may save information in *registers* to be used later by the parser, or to build the representation of the meaning of the sentence. Transformations, e.g., active/passive, can also be handled.
3. **Phrase names**, as well as part-of-speech names, may appear on arcs. This allows a grammar to be called as a subroutine.

The combination of these features gives the ATN the power of a Turing Machine, i.e., it can do anything a computer program can do.

[15] Woods, W. A., "Transition Network Grammars for Natural Language Analysis", *Communications of the ACM*, Oct. 1970

Context Free Parser

A parser for a Context Free language converts a linear string of input tokens into a *parse tree*.

Any program that deals with a tree needs a stack to maintain the list of ancestors of the current node, either as:

- a recursive program: *e.g.*, `expression` parser calls itself as a subroutine
- a program that maintains a *stack*.

These are equivalent: as we shall see, a recursive program is implemented using a runtime stack.

Semantics Influences Parsing

Example: Operator Precedence:

```
        A + B * C
means:  A + (B * C)
not:    (A + B) * C
```

Possible solutions:

1. Unambiguous operator precedence grammar.
   Large, ugly grammar and parser.
   In Pascal, the grammar is not large, but lack of precedence forces the programmer to parenthesize:

   ```
   if x > y and y > z then ...
   ```

   generates errors; it must be written as:

   ```
   if (x > y) and (y > z) then ...
   ```

2. Ambiguous grammar; *precedence* guides parser.
   Short, simple, clean grammar and parser.

Arithmetic Expressions

Example: `(A + B) * C + D`

Ambiguous grammar:

$$
\begin{aligned}
E &\rightarrow \texttt{identifier} \mid \texttt{number} \\
OP &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
E &\rightarrow E \;\; OP \;\; E \\
E &\rightarrow \texttt{(} \; E \; \texttt{)}
\end{aligned}
$$

Unambiguous grammar:

$$
\begin{aligned}
E &\rightarrow E \texttt{+} T \mid E \texttt{-} T \\
E &\rightarrow T \\
T &\rightarrow T \texttt{*} F \mid T \texttt{/} F \\
T &\rightarrow F \\
F &\rightarrow \texttt{(} \; E \; \texttt{)} \\
F &\rightarrow \texttt{identifier} \mid \texttt{number}
\end{aligned}
$$

E, T, F stand for *expression*, *term*, and *factor*.

Example of Operator Precedence

An operator precedence parser examines the current operator and the preceding operator on the stack to decide whether to *shift* the current operator onto the stack or to *reduce* (group) the preceding operator and its operands.

```
1 2 3 4 5 6 7 8
A + B * C + D
```

Pos	Operand Stack	Operator Stack
1	A	
2	A	+
3	A B	+
4	A B	+ *
5	A B C	+ *
6	A (* B C)	+
	(+ A (* B C))	
	(+ A (* B C))	+
7	(+ A (* B C)) D	+
8	(+ (+ A (* B C)) D)	

Operator Precedence

Expressions could be written in an unambiguous, fully parenthesized form. However, this is less convenient for the programmer.

Precedence specifies which operations in a flat expression are to be performed first. `B * C` is performed first in `A + B * C`; `*` *takes precedence over* `+`, `*` $\succ$ `+`.

Associativity specifies which operations are to be performed first when adjacent operators have the same precedence. `A + B` is performed first in `A + B + C` since `+` is *left-associative*. `B ** C` is performed first in `A ** B ** C` since `**` is *right-associative*.

Typical precedence values [*not* Pascal]:

```
10   .                    (highest precedence)
 9   ^
 8   - (unary)
 7   *  /
 6   +  -
 5   =  <>  >=  >  <=  <
 4   not
 3   and
 2   or
 1   :=                   (lowest precedence)
```

Operator Precedence Parsing

Operator precedence parsing is easily done using auxiliary stacks for operands and operators. Tokens are read and processed as follows:

- *Operands:* push onto the operand stack.
- *Operators:*
  While $prec(top-of-stack) \geq prec(op)$, **reduce**.
  Push *op* onto the operator stack.
- `(`: Push `(` onto the operator stack;
  $prec($ `(` $) < prec(any\ operator)$.
- `)`: While $top-of-stack \neq$ `(`, **reduce**.
  Then discard both parentheses.
- *End:* While operator stack is not empty, **reduce**.
  Result is top of operand stack.

reduce combines the two top operands and the top operator into a subtree, which is pushed onto the operand stack.

The $=$ part of the $\geq$ test for operators implements left associativity, the usual case. For right associativity, use $>$ instead.

Operator Precedence Parser

```
(defun expr (inp)           ; opprecc.lsp
  (let (token *op-stack* *opnd-stack*)
    (while inp
      (setq token (pop inp))
      (if (consp token)          ; (exp)
          (push (expr token) *opnd-stack*)
          (if (operatorp token)
              (progn
                (while
                  (>= (prec (first *op-stack*))
                              (prec token))
                  (reducex))
                (push token *op-stack*))
              (push token *opnd-stack*))))
    (while *op-stack* (reducex))
    (pop *opnd-stack*) ))

; Reduce top of stacks to operand
(defun reducex ()
  (let ((rhs (pop *opnd-stack*)))
    (push (list (pop *op-stack*)    ; op
                (pop *opnd-stack*)  ; lhs
                rhs)                ; rhs
          *opnd-stack*) ))
```

Examples

```
(expr '(a + b) )

  ==>  (+ A B)
```

```
(expr '(x := a + b * c) )

  ==>  (:= X (+ A (* B C)))
```

```
(expr '(x := a * b + c) )

  ==>  (:= X (+ (* A B) C))
```

```
(expr '(x := (a + b) * (c + d * e) + f) )

  ==>  (:= X (+ (* (+ A B) (+ C (* D E))) F))
```

Stack Handling in C

- **Initialize** a stack `s` to Empty:

```
s = NULL;
```

- **Test** if stack `s` is not Empty:

```
if ( s != NULL ) ...
```

- **Push** an item `newtok` onto stack `s`:

```
newtok->link = s;
s = newtok;
```

  or:

```
s = cons(newtok,s);
```

- **Pop** stack `s` to yield item `top`:

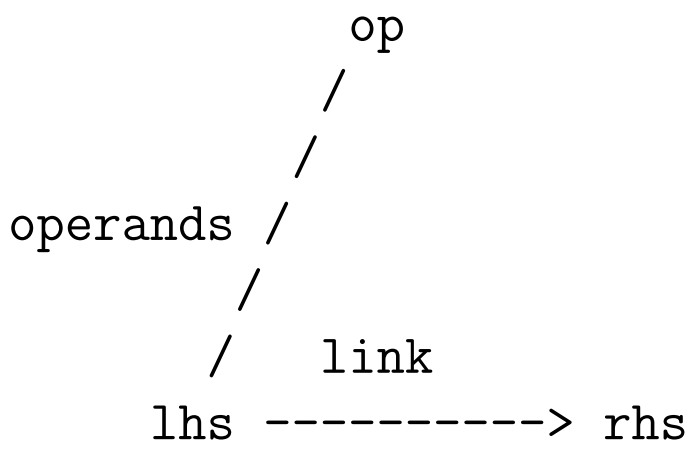
```
top = s;
s = s->link;   /* s = rest(s) */
```

Basic Routines

```
TOKEN opstack, opndstack;

    /*                  +  -  *  /  ... */
int opprec[20] = { 0, 6, 6, 7, 7, ...};

void pushop (TOKEN tok)       /* push op onto stack
  {  tok->link = opstack;
     opstack = tok;  }

TOKEN popop ()                /* pop op from stack *
  { TOKEN tok;
    tok = opstack;
    opstack = tok->link;
    return(tok);   }

int prec (TOKEN tok)         /* precedence of op tok
  { if (tok == NULL)
       return(-1);           /* -1 for empty stack *
       else if (tok->tokentype == OPERATOR)
          return(opprec[tok->whichval]);
          else return(-1);  }  /* -1 for ( */
```

```
reduceop ()          /* reduce binary op   */
  { TOKEN op, lhs, rhs;
    rhs = popopnd();  /* rhs at top */
    lhs = popopnd();
    op = popop();
    op->operands = lhs; /* first child */
    lhs->link = rhs;    /* next sibling */
    rhs->link = NULL;   /* null terminate */
    pushopnd(op);  } /* subtree now operand */
```

We use the *first child - next sibling* form of tree; this represents an arbitrary tree using only two pointers. The tree form of a binary operator and operands is:



Down arrows are always `operands`; side arrows are always `link`. The pretty-printer will print this as:

```
(op lhs rhs)
```

Operator Precedence Parser

```
TOKEN expr ()
{ int done;
  TOKEN tok;
  done = 0;
  opstack = NULL;
  opndstack = NULL;
  while (done == 0)
   { tok = gettoken();
     if (EOFFLG == 0)
        switch (tok->tokentype)
         { case IDENTIFIERTOK: case NUMBERTOK:
             pushopnd (tok);   break;
           case DELIMITER:
             if (tok->whichval == LPARENTHESIS)
               pushop(tok);
             else if (tok->whichval
                        == RPARENTHESIS)
             { while (opstack->whichval
                          != LPARENTHESIS)
                 reduceop();
               popop(); }
             else done = 1;
             break;
```

```
            case RESERVED:
              done = 1;
              break;
            case OPERATOR:
              while (prec(tok) <= prec(opstack))
                 reduceop();
              pushop(tok);
              break;
         }
      else done = 1;
      }
while (opstack != NULL) reduceop();
return (opndstack);   }
```

Additional Considerations

1. Error checking. A variable that keeps track of the type of the previous token is helpful.
   - Previous token was (
   - Previous token was an operator
   - Previous token was an operand
2. Unary operators, such as unary minus. In the proper context, minus can be converted into a different operator that represents unary-minus.
3. Right-associative operators. Shift when the top-of-stack has the same precedence if the operator is right-associative.
4. Type checking and coercion.
5. Function calls. These can be treated like parenthesized subexpressions, keeping track of the number of arguments.
6. Array subscripts. These can be treated like function calls, then modified to perform subscript calculations.

Recursive Descent Parser

A parser for some context-free languages can be written using the technique of *recursive descent*. The basic idea is to write a procedure to parse each kind of statement or expression in the grammar. When such procedures are written in a recursive language, they can call each other as needed.

Example:

`if` *expression* `then` *statement* `else` *statement*

Recursive descent works well for a well-structured language such as Pascal. In Pascal, each statement other than assignment statements begins with a unique reserved word; thus, it is easy for a "big switch" program to determine which statement parser to call.

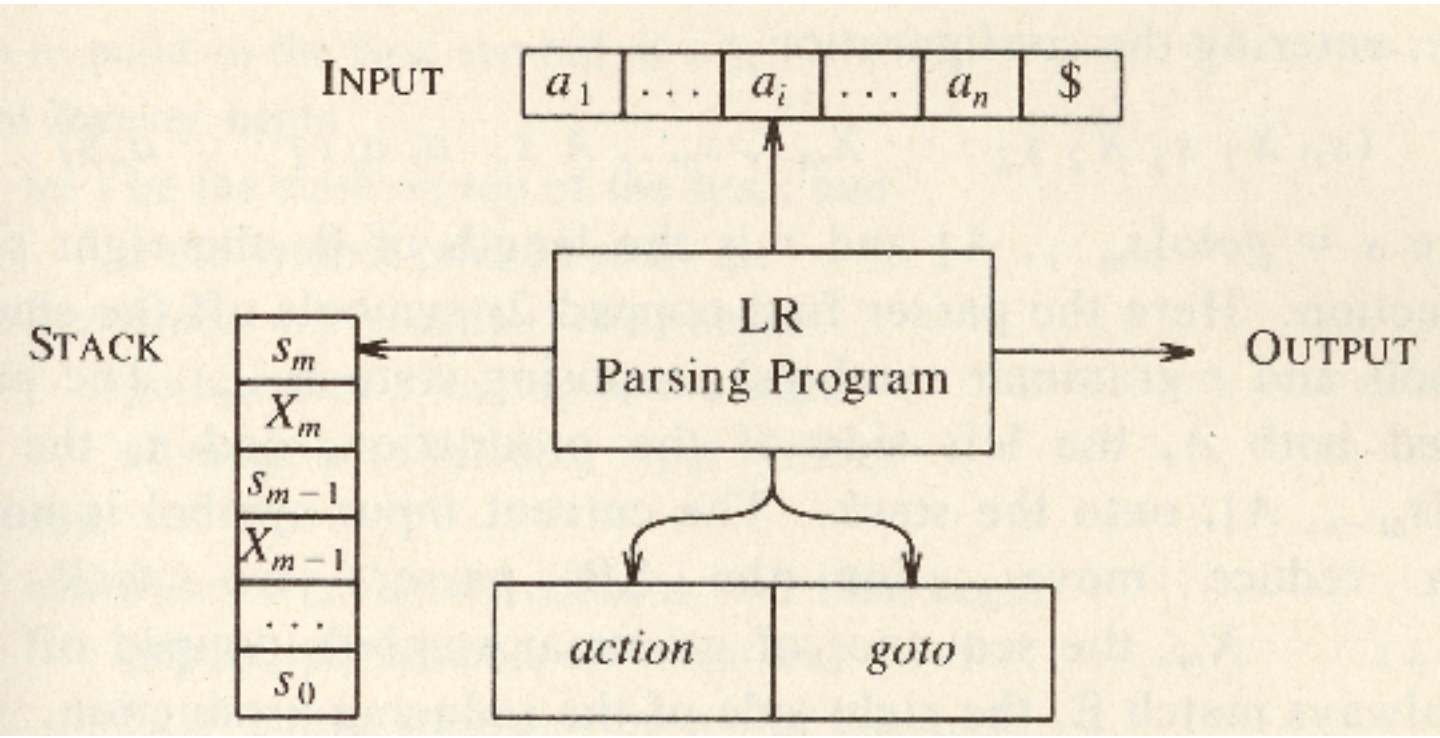
It may be necessary to restructure a grammar to avoid *left recursion*, which can cause a recursive descent parser to go into a loop.

Operator precedence can be used for arithmetic expressions.

Bottom-up Table-driven (LR) Parsing

The most general form of this technique can be used for any language which can be recognized by a deterministic push down automaton.

As one might expect, the parser consists of a stack and a state machine which is derived from the grammar. The overall model is the following:[16]



[16] Aho, Lam, Sethi, and Ullman, *Compilers*, Fig. 4.35.

The LR Parsing Algorithm[17]

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

$(S_0X_1S_1X_2S_2...X_mS_m, a_ja_{j+1}...a_n\$)$

This configuration represents the right-sentential form:

$(X_1X_2...X_ma_ja_{j+1}...a_n)$

The next move of the parser is determined by reading a_j, the current input symbol, and S_m, the state on top of the stack, and then consulting the parsing action table entry, $action[S_m, a_j]$.

[17] slide by John Werth

Shift-Reduce Parser

Based on the current state and current input, $action[S_m, a_j]$ gives one of four possible actions:

- **Shift** s**:** Shift the current input a_j onto the stack and go to the specified next state s.
- **Reduce** r**:** Given the production $r : A \rightarrow \beta$, remove enough items from the stack to match β, produce output structure from those items, and put the resulting structure A back on the stack. The next state is found from the *goto* table.
- **Accept:** Parsing is completed; return top of stack as result.
- **Error:** The input cannot be parsed according to the grammar; call an error recovery routine. Error entries are denoted by blanks in the parsing table.

Example Parsing Table[18]

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

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

[18] Aho, Lam, Sethi, and Ullman, *Compilers*, Fig. 4.37, p. 252.

A Parse of `id * id + id`[19]

	STACK	INPUT	ACTION
(1)	0	**id** ∗ **id** + **id** $	shift
(2)	0 **id** 5	∗ **id** + **id** $	reduce by $F \rightarrow$ **id**
(3)	0 F 3	∗ **id** + **id** $	reduce by $T \rightarrow F$
(4)	0 T 2	∗ **id** + **id** $	shift
(5)	0 T 2 ∗ 7	**id** + **id** $	shift
(6)	0 T 2 ∗ 7 **id** 5	+ **id** $	reduce by $F \rightarrow$ **id**
(7)	0 T 2 ∗ 7 F 10	+ **id** $	reduce by $T \rightarrow T*F$
(8)	0 T 2	+ **id** $	reduce by $E \rightarrow T$
(9)	0 E 1	+ **id** $	shift
(10)	0 E 1 + 6	**id** $	shift
(11)	0 E 1 + 6 **id** 5	$	reduce by $F \rightarrow$ **id**
(12)	0 E 1 + 6 F 3	$	reduce by $T \rightarrow F$
(13)	0 E 1 + 6 T 9	$	$E \rightarrow E+T$
(14)	0 E 1	$	accept

[19] Aho, Lam, Sethi, and Ullman, *Compilers*, Fig. 4.38, p. 253.

Synthesized Translation

The term *synthesized translation* includes two ideas:

- A complex expression is constructed from subexpressions.

  `if` *expression* `then` *statement*

- The translation of a complex expression is constructed in a mechanical way from the translations of its subexpressions.

In the statement `if x > max then max := x` the code for the `if` statment is constructed by linking together the code that is produced for the subexpressions `x > max` and `max := x`.

```
   >              :=
  /              /
 /              /
x ---- max     max ---- x
```

```
     if            (if (> x max) (:= max x))
     /
    /
   > ----------- :=
  /              /
 /              /
x ---- max     max ---- x
```

Using `yacc`

`yacc` compiles an input file (with `.y` suffix), always producing the output file `y.tab.c` (a C file that can be compiled to an executable using the C compiler).

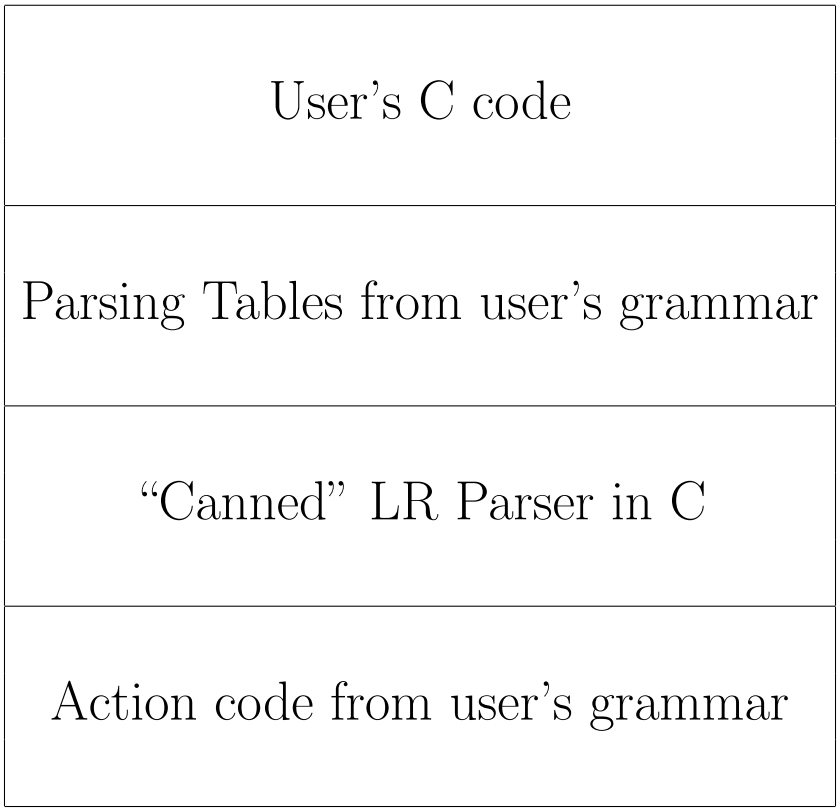
`y.tab.c` contains:

- a table-driven parser,
- tables compiled from the input file's grammar rules,
- the C code from the input file.

The parser in `y.tab.c` is designed to call a lexical analyzer produced by `lex`. The user's C code will contain the `main()` program; `main()` calls the code generator as a subroutine after parsing is complete.

y.tab.c

The file `y.tab.c` produced by YACC has the following structure:



Yacc Specifications[20]

```
%{  declarations  %}
tokens
%%
translation rules
%%
auxiliary procedures
```

- declarations: `#include` and manifest constants (identifier declared to represent a constant).
- tokens: `%token` declarations used in the second section. Each token is defined as a constant (integer > 255).
- translation rules: pattern / action pairs
- auxiliary procedures: arbitrary C functions copied directly into the generated lexical analyzer.

[20] slide by John Werth.

Example: Desk Calculator

```
%{ /* simcalc.y  -- Simple Desk Calculator    */
   /* Aho, Sethi & Ullman, Compilers, Fig. 4.56 */
#include <ctype.h>
#include <stdio.h>
%}
%token DIGIT
%%
line   :  expr '\n'          { printf("%d\n", $1); }
       ;
expr   :  expr '+' term      { $$ = $1 + $3; }
       |  term
       ;
term   :  term '*' factor    { $$ = $1 * $3; }
       |  factor
       ;
factor:   '(' expr ')'       { $$ = $2; }
       |  DIGIT
       ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
       yylval = c - '0' ;
       return DIGIT;
    }
    return c;
  }
```

Yacc: Pascal Subset

```
program     : statement DOT     /* change this! */
                { parseresult = $1; }  ;
statement   : BEGINBEGIN statement endpart
                { $$ = makeprogn($1,cons($2,$3)); }
            | IF expr THEN statement endif
                { $$ = makeif($1, $2, $4, $5); }
            | assignment  ;
endpart     : SEMICOLON statement endpart
                    { $$ = cons($2, $3); }
            | END  { $$ = NULL; }  ;
endif       : ELSE statement  { $$ = $2; }
            | /* empty */     { $$ = NULL; }  ;
assignment  : IDENTIFIER ASSIGN expr
                { $$ = binop($2, $1, $3); }  ;
expr        : expr PLUS term
                { $$ = binop($2, $1, $3); }
            | term  ;
term        : term TIMES factor
                { $$ = binop($2, $1, $3); }
            | factor  ;
factor      : LPAREN expr RPAREN  { $$ = $2; }
            | IDENTIFIER
            | NUMBER  ;
```

Auxiliary C Code

```
TOKEN cons(item, list)  /* link item to list */
  TOKEN item, list;
  { item->link = list;
    return item;
  }

TOKEN binop(op, lhs, rhs)  /* reduce binary op */
  TOKEN op, lhs, rhs;
  { op->operands = lhs;    /* link opnds to op */
    lhs->link = rhs;       /* link 2nd operand */
    rhs->link = NULL;      /* terminate opnds  */
    return op;
  }

TOKEN makeprogn(tok, statements) /* make progn */
  TOKEN tok, statements;
  {  tok->tokentype = OPERATOR;  /* change tok */
     tok->whichval = PROGNOP;    /* to progn*/
     tok->operands = statements;
     return tok;
   }
```

Auxiliary C Code ...

```
TOKEN makeif(tok, exp, thenpart, elsepart)
  TOKEN tok, exp, thenpart, elsepart;
  {  tok->tokentype = OPERATOR;  /* change tok */
     tok->whichval = IFOP;       /*   to if op */
     if (elsepart != NULL)
         elsepart->link = NULL;
     thenpart->link = elsepart;
     exp->link = thenpart;
     tok->operands = exp;
     return tok;
   }
```

Controllability and Observability

These are central concepts from control theory. We will define them as:

- **Controllability:** the ability to change the behavior of a system by changing its parameters.
- **Observability:** the ability to observe the behavior of a system well enough to control it.

In order to control a system, both controllability and observability are required.

The implications for large software systems are:

- Aspects of software that cannot easily be observed will never be debugged.
- All large software systems must have observability built in.
- Observability is a requirement, not a luxury. The time spent building in observability will be well repaid.
- In-process traces can be turned on by setting bits in a bit vector.

Example

```
i:=j.    /* input */

binop
 79220  OP      :=  link     0  operands 79172
 79172  ID       I  link 79268
 79268  ID       J  link     0

yyparse result = 0
 79220  OP      :=  link     0  operands 79172

(:= I J)
```

Examples ...

```
begin i:=j; j:=7 end.    /* input */
binop
 79460  OP     :=  link     0  operands 79412
 79412  ID      I  link 79508
 79508  ID      J  link     0
binop
 79652  OP     :=  link     0  operands 79604
 79604  ID      J  link 79700
 79700  NUM     7  link     0
cons
 79652  OP     :=  link     0  operands 79604
     0  NULL
cons
 79460  OP     :=  link 79652  operands 79412
 79652  OP     :=  link     0  operands 79604
makeprogn
 79364  OP  progn  link     0  operands 79460
 79460  OP     :=  link 79652  operands 79412

yyparse result = 0
 79364  OP  progn  link     0  operands 79460

(progn (:= I J)
       (:= J 7))
```

Examples ...

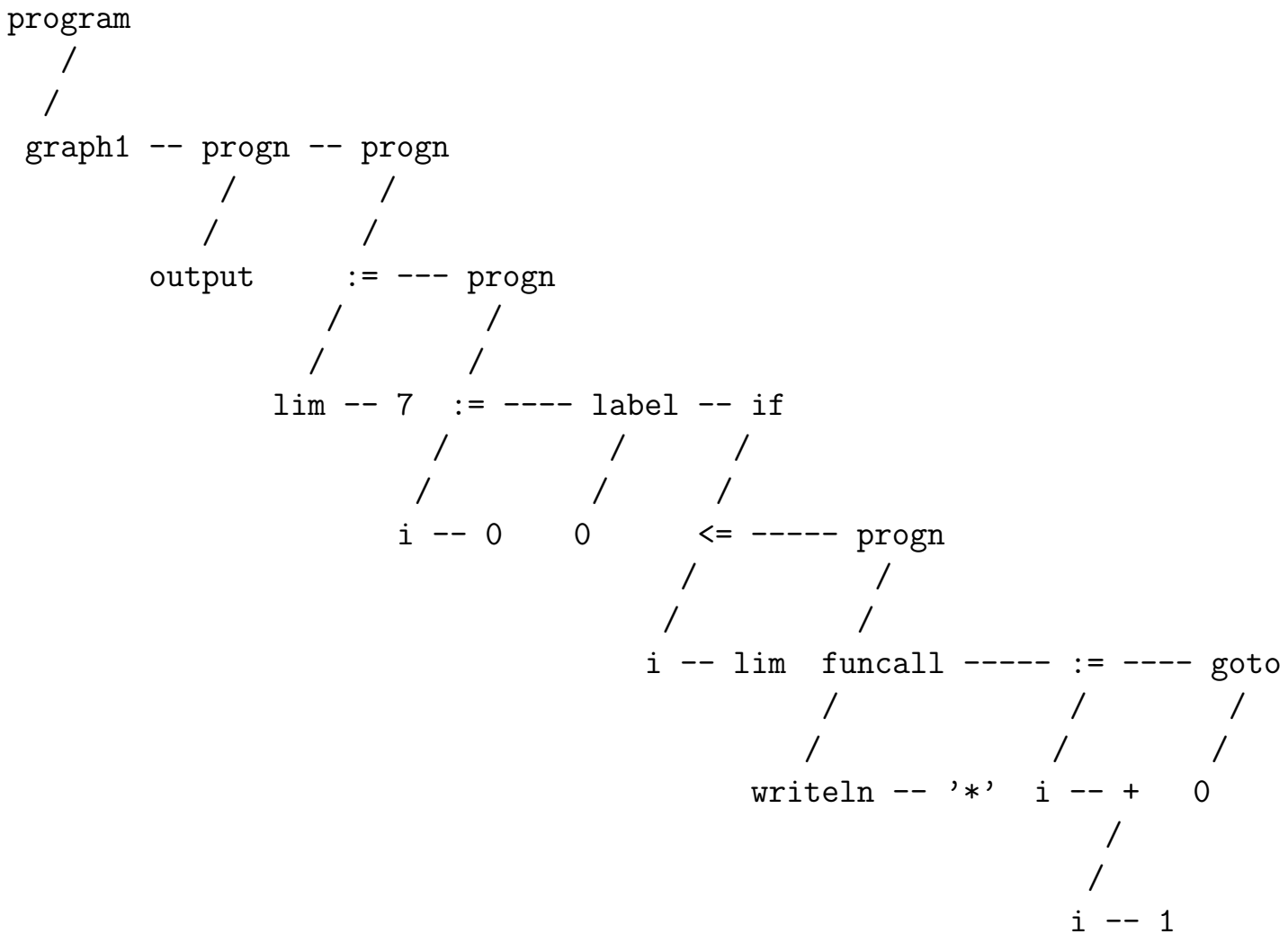
```
if i+j then begin i:=j; j:=3 end else k:=i .

binop 79940  OP      +,  ID      I,  ID      J
binop 80180  OP     :=,  ID      I,  ID      J
binop 80372  OP     :=,  ID      J,  NUM     3
cons  80372  OP     :=,      0  NULL
cons  80180  OP     :=,  80372  OP     :=
makeprogn
 80084  OP  progn,  80180  OP     :=
binop 80612  OP     :=,  ID      K,  ID      I
makeif
 79844  OP     if  link     0  operands 79940
 79940  OP      +  link 80084  operands 79892
 80084  OP  progn  link 80612  operands 80180
 80612  OP     :=  link     0  operands 80564
yyparse result = 0
 79844  OP     if  link     0  operands 79940
(if (+ I J)
    (progn (:= I J)
           (:= J 3))
    (:= K I))
```

Hints for `yacc`

Some useful hints for using `yacc`:

- Avoid "empty" productions; these are likely to generate grammar conflicts that may be hard to find. Each production should consume some input.
- Follow the Pascal grammar flowcharts exactly. If you just write the grammar from your memory of Pascal syntax, it probably won't work.
- When the action code for a production is called, all of the `$i` variables have been completely processed and have values.
- If you need to process a list of items of the same kind, the code for `begin` is a good model.
- The `yacc` stack has a single type; for our program, that type is `TOKEN`. If you want to return something else (e.g. a `SYMBOL`), package it in a `TOKEN`.

File `trivb.tree`



The Semantic Actions[21]

Semantic actions refer to the elements of the production by a positional notation. `$$` denotes the left side nonterminal. `$n` refers to the nth symbol on the right-hand side.

For example:

```
expr : expr '+' term  { $$ = $1 + $3; }
```

The default action action is `{ $$ = $1; }`. For this reason, we do not need to have an explicit action for

```
expr : term
```

[21] slide by John Werth.

Supporting C Routines[22]

A lexical analyzer named `yylex()` must be provided in the third section. This can be user-written or generated by lex.

`yylex` must return *two* values:

- "what is it?", a small integer that is the return value, e.g. `return(NUMBER);`
- the actual value of the item, by setting the value of `yylval`, e.g. `yylval = tok;`

[22]adapted from slide by John Werth.

Example

```
%{  /* Aho, Lam, Sethi, Ullman, Compilers: Fig. 4.59 */
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double   /* double type for yacc stack */
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines:  lines expr '\n'     { printf("%f\n",$2); }
     |  lines '\n'
     |  /* epsilon */
     |  error '\n'          {yyerror("reenter last line:");
                             yyerrok; }
     ;
expr:   expr '+' expr       { $$ = $1 + $3; }
    |   expr '-' expr       { $$ = $1 - $3; }
    |   expr '*' expr       { $$ = $1 * $3; }
    |   expr '/' expr       { $$ = $1 / $3; }
    |   '(' expr ')'        { $$ = $2; }
    |   '-' expr %prec UMINUS  { $$ = - $2; }
    |   NUMBER
    ;
%%
yylex() {
  int c;
  while ( (c = getchar()) == ' ');
  if ( (c == '.') || (isdigit(c)) ) {
     ungetc(c, stdin);
     scanf("%lf", &yylval);
     return NUMBER;
     }
  return c;  }

yyerror(s)
  char * s;
  { fputs(s,stderr), putc('\n',stderr);  }

main()          /* Call repeatedly to test */
  { int res;
    while (1) { res = yyparse(); } }
```

Comments on the Example[23]

The statement `#define YYSTYPE double` sets `YYSTYPE`, the type of the yacc semantic stack. This stack parallels the syntax stack maintained by the LALR parser. References like `$1` to values in the the right hand side of a production are actually references to this stack of values. This stack could have record type or union type; the latter is typical for a compiler.

Note that a grammar may be ambiguous, with parsing action conflicts; yacc will report this.

To see the sets of items and the specific parsing action conflicts, use the `yacc -v` option. The diagnostic information will be produced in a file called `y.output` .

[23] slide by John Werth.

Parsing Action Conflicts[24]

Parsing action conflicts are not necessarily fatal; yacc resolves them using the following default rules:

- reduce/reduce is resolved by choosing the first listed production.
- shift/reduce is resolved in favor of shift.

A more general mechanism used to resolve shift reduce is to associate a precedence and associativity to both productions and terminal.

Precedence of terminal depends on the order in which it appears in the specification: for example, `*` is higher than `+` in the earlier example.

The precedence and associativity of a production is that of the rightmost terminal by default. It can be set with `%prec` to that of any terminal. For example,

```
expr : '-' expr %prec UMINUS
```

sets the production associativity and precedence to that of `UMINUS`.

[24]slide by John Werth.

Resolving Shift/Reduce Conflicts[25]

The rule for reducing shift/reduce conflicts is:

- **reduce** if the precedence of the production is greater than that of the terminal or if they are equal and the associativity of the production is left.
- **shift** otherwise.

Our Pascal grammar has a shift/reduce conflict in the case of the `if` statement; these rules resolve it in favor of shift, which does the right thing.

[25] slide by John Werth.

Error Productions[26]

The earlier example grammar contains an error production:

```
lines : error '\n' { yyerror("reenter last line:");
                     yyerrok; }
```

An error production uses the keyword `error` followed by a string: `A : error` α
`yyerrok` resets the parser so it can continue.

[26] slide by John Werth.

Error Handling[27]

During the parse, if an error is encountered (access to a blank entry in the LALR table) the following happens:

1. the stack is popped until a state is on top whose items match the right-hand side of an `error` production.
2. a token `error` is shifted onto the stack
3. if α is empty, then a reduction takes place and the semantic action is performed.
4. if α is non-empty, the parser tries to identify α in the input. It then pushes α on the stack and does a reduction as usual.

For example, `stmt : error ';'` will have the effect of causing the parser to skip ahead to the next `;` and assume that a statement has been found.

[27] slide by John Werth.

Parsing Techniques

Recursive Descent and Operator Precedence:

- **Advantages:**
  - Easy to control, understand.
  - Good error messages can be produced.
- **Disadvantages:**
  - More code to write.
  - Grammar not as clear.
    Does the program really match the grammar?

Parser Generator:

- **Advantages:**
  - Less code to write.
  - Grammar is clear and easy to change.
- **Disadvantages:**
  - The programmer may get "stuck" on grammar errors that are hard to understand.
  - Error messages during compilation may be unhelpful and far from the actual error.

Looping Statements

Looping statements generate multiple statements in intermediate code, as shown in the following patterns. (Generate label numbers by `j = labelnumber++;`.)

```
for i := start to end do s

(PROGN (:= i start)
       (LABEL j)
       (IF (<= i end)
           (PROGN s
                  (:= i (+ 1 i))
                  (GOTO j) )))
while c do s

(PROGN (LABEL j)
       (IF c (PROGN s (GOTO j))))


repeat statements until c

(PROGN (LABEL j)
       (PROGN statements)
       (IF c (PROGN) (GOTO j)))
```

The empty `(PROGN)` acts as a no-op.

Symbol Table

A *symbol table* is a data structure that associates *names* with *information about the objects* that are denoted by the names.

Programming languages have many kinds of symbols: statement labels, variables, constants, subprogram names, etc.

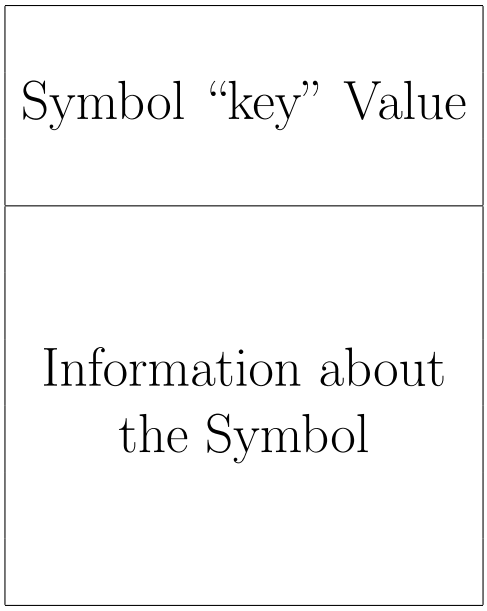
A symbol table must be well organized:

- For fast lookup of symbols.
  Note that if symbol table lookup takes $O(n)$ time, the total compilation time becomes $O(n^2)$.
- To reflect the organization of the program (block structure).

A set of symbol table routines for use with the class programs is provided in the file `symtab.c` and documented in the file `symtab.txt` .

Symbol Table Organization

A *symbol table* is a set of data structures and associated procedures that allow "symbols" (perhaps defined rather broadly) and associated information to be stored and looked up.



Operations:

- **Insert:** Insert a new symbol into the table.
- **Search:** Search for a symbol in the table.
- **Search/Insert:** Search for a symbol, Insert if not found.

We are often interested in the performance (expected time) of these operations. There are more searches than insertions.

Symbol Table Organizations

Linear, Unsorted:

Symbols are stored in an array; new symbols are inserted at the end.

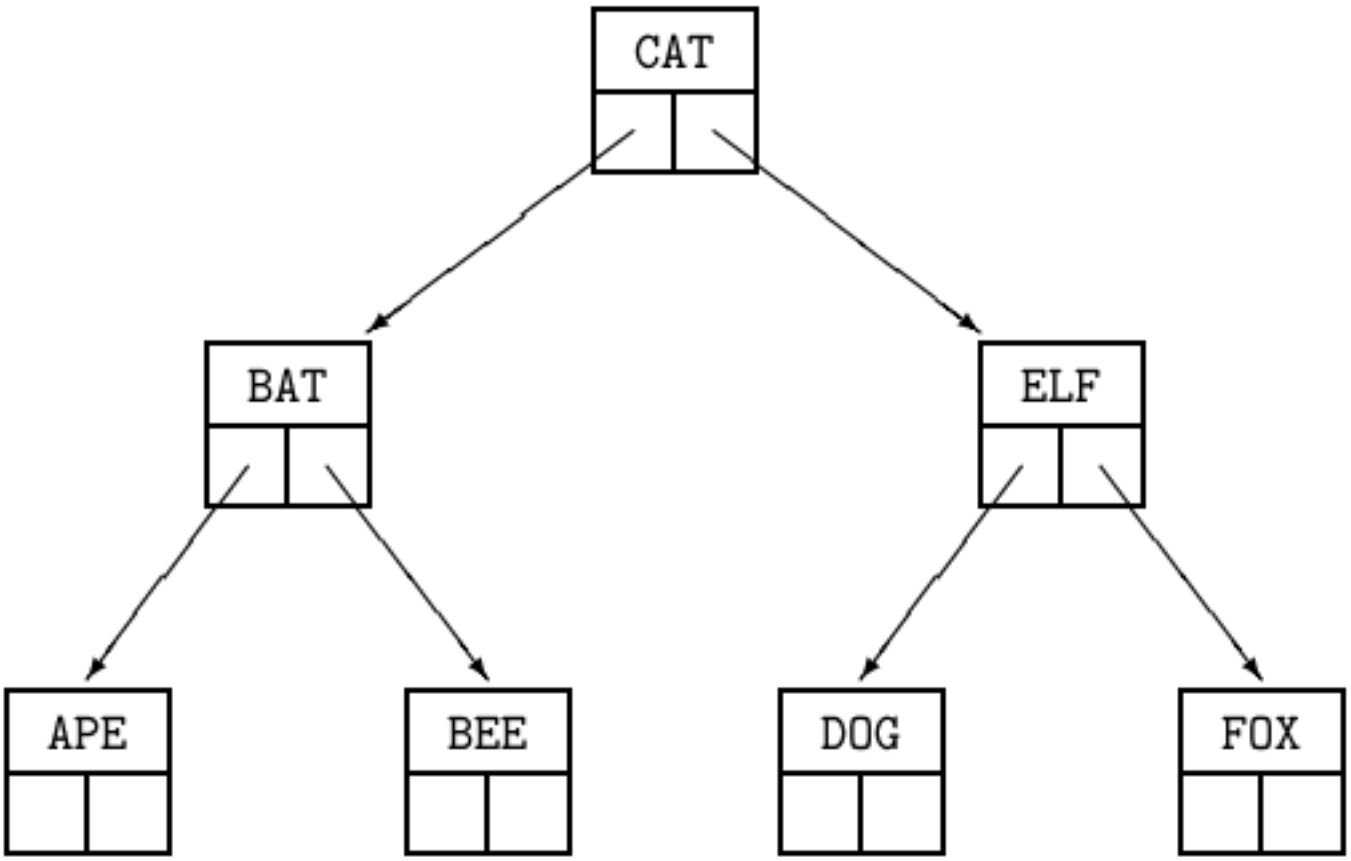
Insert: $O(1)$
Search: $O(n)$ Search half the table on average.

Linear, Sorted:

Symbols are stored in an array that is sorted (*e.g.*, alphabetically).

Insert: $O(n)$ Move half the table on average.
Search: $O(log\ n)$ Binary search.
Good for reserved words (no insertions).

Binary Tree Symbol Table



Compare search key to contents of node:

- = Found
- < Search left subtree
- > Search right subtree

Advantage: $O(log\ n)$ search time if tree is balanced.

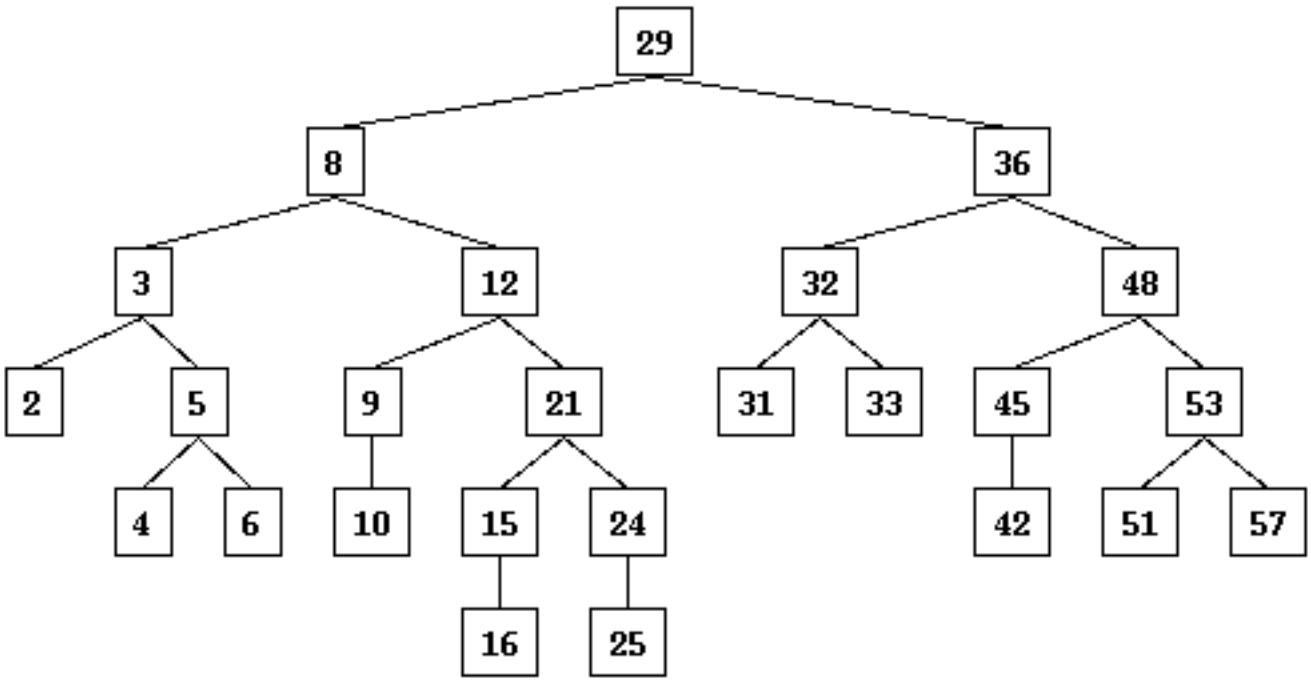
Disadvantage: Search time may be long if tree is not balanced.

AVL Trees

An AVL Tree [28] is a binary tree that is approximately height-balanced: left and right subtrees of any node differ in height by at most 1.

Advantage: Approximately $O(log\ n)$ search and insert time.

Disadvantage: Somewhat complex code (120 - 200 lines).



[28] G. M. Adel'son-Vel'skiĭ and E. M. Landis, *Soviet Math.* **3**, 1259-1263, 1962; D. Knuth, *The Art of Computer Programming, vol. 3: Sorting and Searching*, Addison-Wesley, 1973, section 6.2.3.

Hash Table

Symbols are stored in a *hash table*, indexed by a *hash function* applied to the key value (*e.g.*, multiply a symbol by itself and take some bits from the middle of the product). The hash function always produces the same value for a given symbol, but is otherwise "random". If there is a *collision* (two different symbols hash to the same value), it is necessary to *rehash* (*e.g.*, try the next place in the table).

Insert:	$O(1)$	Depends on table fullness
Search:	$O(1)$	Depends on table fullness

The expected time is only 5 comparisons for a table that is 90% full; time depends on fullness but not on table size.

Advantages:

- Fast
- Works for symbols without a natural sort order

Disdvantages:

- More complex code
- Have to find a good hashing function
- Must dedicate a large table, or rehash the table to expand.

Hash Functions

Not all hash functions are good; a good one should be easy to compute and should "randomize" symbols into different hash locations, not lump them into only a few hash values.

Some examples of good hash functions:

- Treat the symbol name as an integer, square it, and take some bits out of the middle.
- Treat the symbol name as an integer and find the remainder of the name modulo `p`, where `p` is prime.

A typical re-hash function is simply to add 1 to the previous hash function value.

Indexed Buckets

A symbol is used to index a table, which provides a pointer to a linked list of symbols.

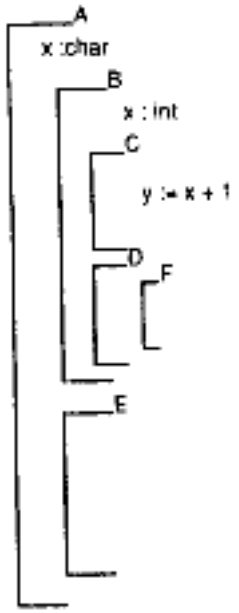
Hashed with Buckets:
A hashing function is used to index the table of buckets.

Insert: $O(1)$
Search: $O(n)$ Actually $n/(2 \cdot n_b)$
n_b (number of buckets) can be large.
Since pointers are used for the linked list of symbols, the table can be expanded by requesting more memory.

Alphabetized Buckets:
The first character of first few characters of a symbol are used to index the table of buckets.

Insert: $O(n)$ Linked list is sorted
Search: $O(n)$ Actually $n/(2 \cdot n_b)$
Some letter combinations are much more common, so some bucket entries will be empty. Symbols are in a sorted order.

Lexical Scoping[29]

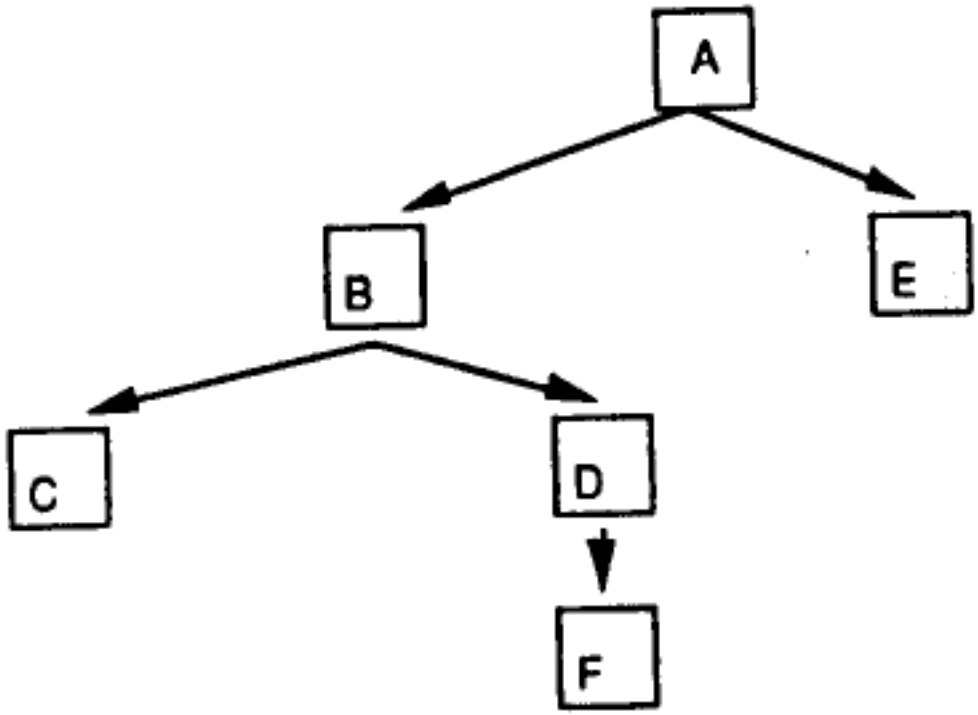


Scope is the region of program text over which a symbol can be referenced. In most languages, the scope of a variable is determined by syntax, e.g. a variable can be used only inside the procedure that defines it. This notion of scope is reflected in the symbol table, which is called *lexical scoping*. With *lexical scoping*, a name is defined when it can be looked up in the lexicon (symbol table) at the point of use.

Some languages (Algol, Pascal, PL/I) have allowed procedures to be nested inside other procedures; however, this has a high runtime cost and only slight benefit.

[29]adapted from slide by John Werth.

Tree of Symbol Tables[30]



The nested procedures of the previous figure can be thought of as a tree. Each procedure can access its symbols and the symbols of its ancestors. A symbol at a lower level *shadows* any symbols with the same name at higher levels, e.g. in method overriding in OOP.

Our symbol table will have only two levels:

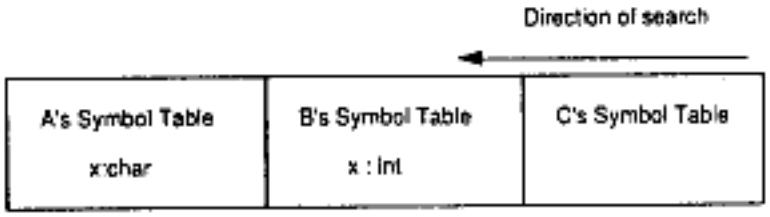
- Level 0: predefined symbols such as `integer`
- Level 1: symbols of the program being compiled

[30] adapted from slide by John Werth.

Stack Symbol Table[31]

During compilation, the compiler performs a depth-first traversal of the tree of procedures.

Since a procedure can access its symbols and the symbols of its ancestors, the symbol table could be structured as a stack.



Searching backwards through the stack implements *shadowing* , i.e. the closest definition of a symbol with a given name takes precedence.

When parsing of a procedure is finished, its local symbol table can be popped off the stack.

Alternatively, the whole tree of symbol tables can be maintained, with each node having a pointer to its parent.

[31] adapted from slide by John Werth.

Use of Symbol Table

Usually, there are some symbols that are predefined by the language, e.g. `integer`. These will be put into the symbol table by the initialization of the compiler.

User symbols are added in response to *declaration statements*; these statements usually appear at the top of a procedure. Declaration statements often do not generate any code, but cause symbol table entries to be made and affect code generation for executable statements.

Symbols encountered in code, e.g. in assignment statements, are looked up in the symbol table. If a symbol is not found in the table, an *undefined symbol* error is generated. Looking up a symbol provides its type and its memory address (offset), which are needed for type checking and code generation.

Symbol Table Entry

A symbol table entry stores a variety of information about a symbol:

`link`	Link to Next Symbol
`namestring`	Name of Symbol
`kind`	Kind of Symbol: `VARSYM`
`basicdt`	Basic Data Type: `REAL`
`datatype`	Pointer to Data Type
`blocklevel`	Block Nesting Level: `1`
`size`	Size in bytes
`offset`	Offset from Data Area
`intnum, realnum`	Constant Value
`lowbound`	Low Bound
`highbound`	High Bound

Kinds of Symbols

A compiler must typically deal with several different kinds of symbols:

- Symbolic constants: `const pi = 3.1415926`
  Store constant value and type; convert an identifier with this name to a numeric constant as soon as it is looked up. The language may have some built-in symbolic constants, such as `maxint`.
- Enumerated types:
  `type color = (red, white, blue);`
  Simply assign constant values `0`, `1`, ... to the items. The name `color` becomes a subrange `0..2`.
- Subrange types: `type digit = 0..9;`
  Store the associated scalar type and the lower and upper bounds (always integers).
- Scalar types:
  `boolean integer real char pointer`
  These are predefined as types in the symbol table.
- Variables: these may have scalar types or complex types.

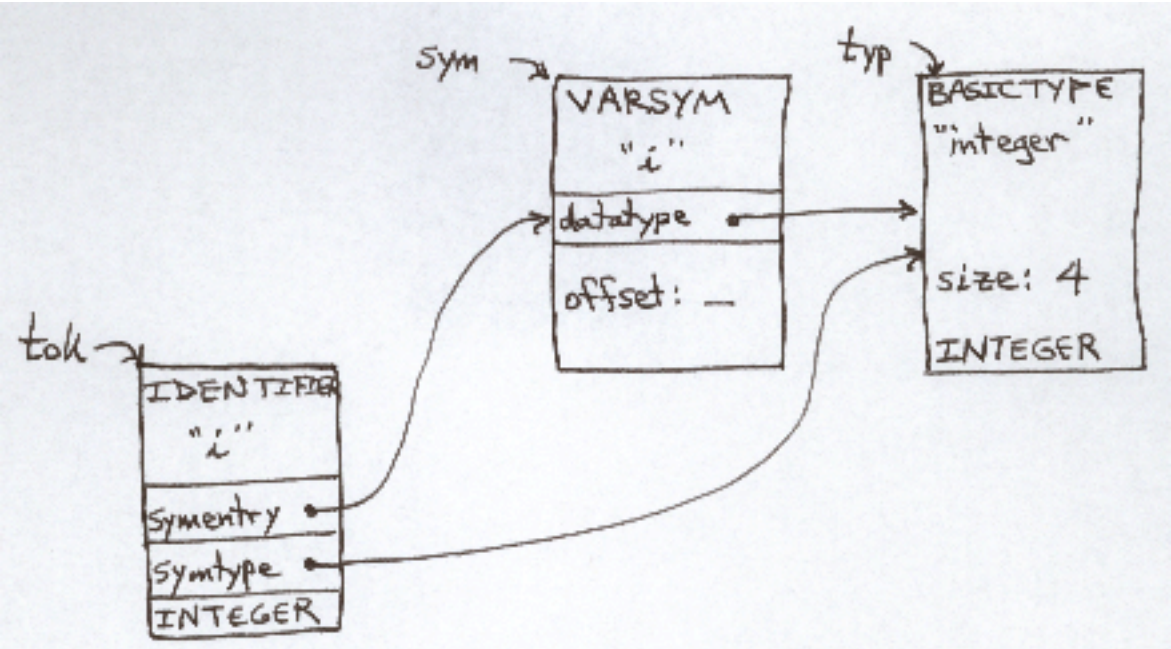
Kinds of Symbols ...

- Arrays: `array[1..10] of date`
  Store the number of dimensions, lower and upper bound for each, type of array element, number of address units per element.
- Records:
  `type vector = record x,y : integer end;`
  Store each element name, type, and offset. Variant records simply restart the offset at the start of the variant part.
- Functions and procedures: Store number and types of arguments, result type. Some functions, such as `abs ord succ`, may be compiled as in-line code.
- Labels: keep in a separate table if they have a form different from symbols.
- Objects: a *class* in OOP packages together a record of data items and a set of methods, which are procedures associated with the class. In OOP, some form of symbol table for the class is output as well as code for the methods.

Looking up ID in Symbol Table

An identifier that is encountered in executable code must be linked to its appropriate symbol table entries:

```
TOKEN findid(TOKEN tok) { /* the ID token */
SYMBOL sym, typ;

sym = searchst(tok->stringval);
tok->symentry = sym;
typ = sym->datatype;
tok->symtype = typ;
if ( typ->kind == BASICTYPE ||
     typ->kind == POINTERSYM)
    tok->datatype = typ->basicdt;
```



Variable Declarations

A variable declaration has a form such as:

$$\texttt{var}\ var_1, var_2, ..., var_n\ \texttt{:}\ type\ \texttt{;}$$

Such a declaration is processed as follows:

1. Find the symbol table entry for *type* .
2. For each variable var_i,
   (a) Allocate storage within the current block using the storage allocation algorithm and the size of *type* .
   (b) Make a symbol table entry for the variable, filling in its print name, type, offset, size, and block level.
   (c) Enter the symbol in the symbol table for the current block.

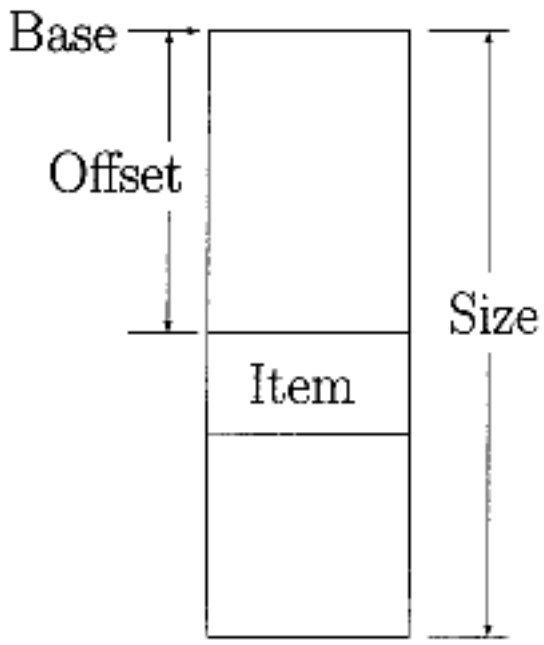
Identifier List etc.

```
idlist    :  IDENTIFIER COMMA idlist
                              { $$ = cons($1, $3); }
          |  IDENTIFIER       { $$ = cons($1, NULL); }
          ;
vblock    :  VAR varspecs block          { $$ = $3; }
          |  block
          ;
varspecs  :  vargroup SEMICOLON varspecs
          |  vargroup SEMICOLON
          ;
vargroup  :  idlist COLON type
                                { instvars($1, $3); }
          ;
type      :  simpletype
          | ... ;
simpletype :  IDENTIFIER   { $$ = findtype($1); }
          | ... ;
```

Data Addressing

A *data area* is a contiguous region of storage specified by its *base address* and *size*.

An item within a data area is specified by the *base address* of the data area and the *offset* of the item from the base address.



Two kinds of data areas are arrays and records.

Note that since an item in a data area may itself be a data area, the layout of data in memory may be considered to be a "flattened tree". A reference to data is a sequence of steps down this tree until the desired data is reached.

Storage Allocation

Allocation of storage is done as an *offset* to a *base address*, which is associated with a block of storage. Assignment of storage locations is done sequentially by a simple algorithm:

- Initially, `next = 0`.
- To allocate an item of size `n`:

```
offset = next;
next = next + n;
return offset;
```

- Finally, `next` gives the total size of the block.

In our compiler, the `next` variable for allocating variables is `blockoffs[blocknumber]`.

Alignment and Padding

Certain items must be allocated at restricted locations; *e.g.*, a floating point number must be allocated at a word (4-byte or 8-byte) boundary. This is called *storage alignment*.

In such cases, `next` is advanced to the next available boundary if needed, and the intervening storage is wasted; this is called *padding*.

To pad to a boundary of size `m` , perform:

```
wordaddress(next, m)
   = ( (next + m - 1) / m) * m
```

using truncating integer arithmetic.

For records, a compaction algorithm could be used to minimize wasted storage.

Installing Variables in Symbol Table

```
/* install variables in symbol table */
void instvars(TOKEN idlist, TOKEN typetok)
  {  SYMBOL sym, typesym; int align;
     typesym = typetok->symtype;
     align = alignsize(typesym);
     while ( idlist != NULL )   /* for each id */
       {  sym = insertsym(idlist->stringval);
          sym->kind = VARSYM;
          sym->offset =
              wordaddress(blockoffs[blocknumber],
                          align);
          sym->size = typesym->size;
          blockoffs[blocknumber] =
                         sym->offset + sym->size;
          sym->datatype = typesym;
          sym->basicdt = typesym->basicdt;
          idlist = idlist->link;
        };
  }
```

`blockoffs[blocknumber]` is the offset in the current block; this is the `next` value for this storage allocation.

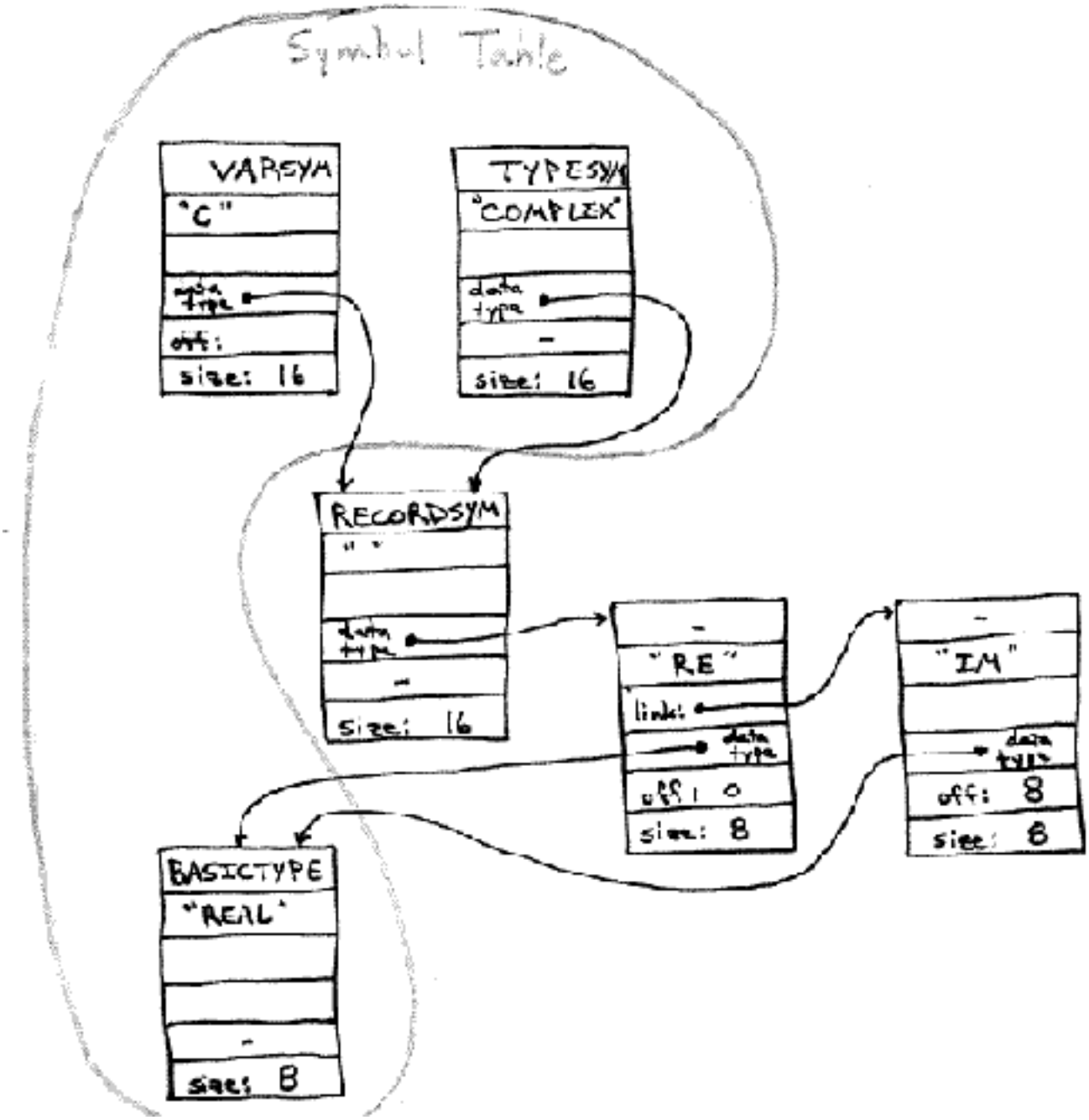
Record Declarations

A record declaration has a form such as:
`record` $field_1, ..., field_n$ `:` $type_1$ `;` ... `end`
Such a declaration is processed as follows:

1. Initialize offset within the record to be `0`.
2. For each entry group,
   (a) Find the symbol table entry for the *type* .
   (b) Allocate storage within the record using the storage allocation algorithm and size of *type* .
   (c) Make a symbol table entry for each field, filling in its print name, type, offset in the record, and size.
   (d) Link the entries for the fields to an entry for the record.
3. The size of the record is the total size given by the storage allocation algorithm, rounded up to whole words, e.g. multiple of 8.
4. Variant records simply restart the storage allocation at the place where the variant part begins. Total size is the maximum size of the variants.

Symbol Table Structures for Record

```
type complex = record re, im: real end;

var c: complex;
```



Array Declarations

A simple array declaration has a form such as:

`array [` $low_1..high_1$ `] of` *type*

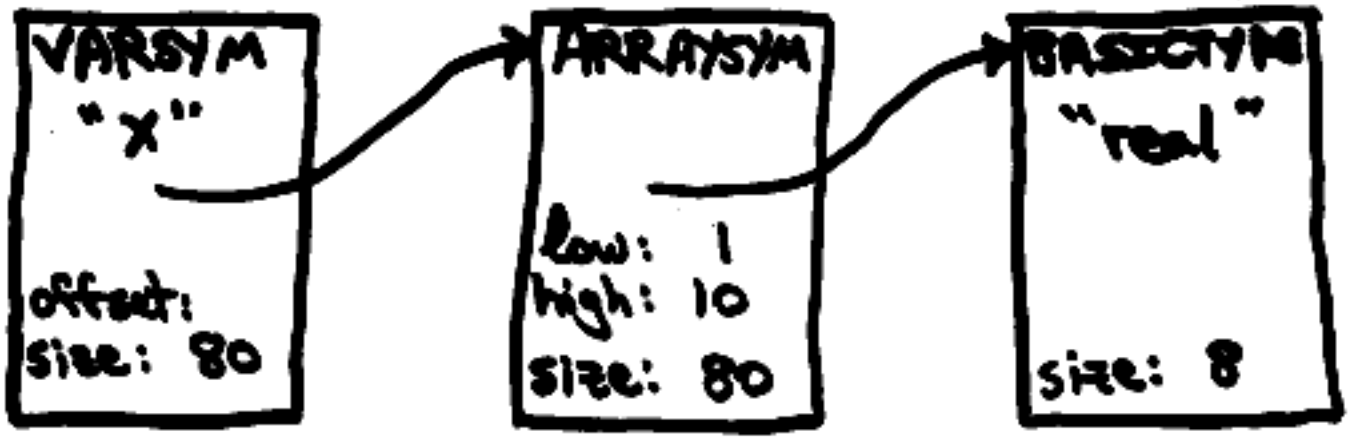
Such a declaration is processed as follows:

1. Find the symbol table entry for *type* .
2. Make a symbol table entry for the array type. The total size of the array is:
   $(high_1 - low_1 + 1) * size(type)$

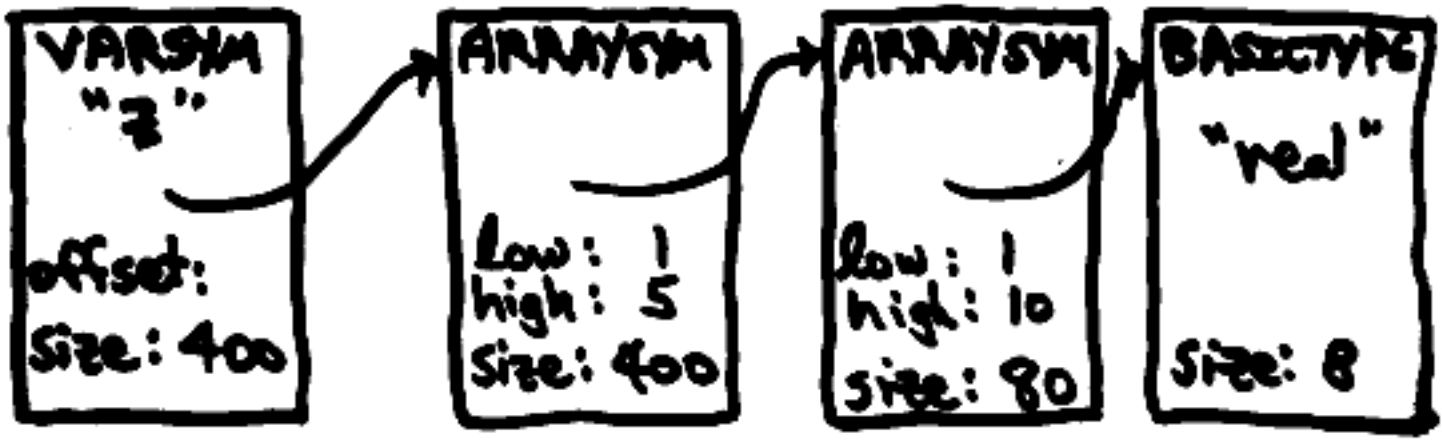
Multiply dimensioned arrays can be treated as arrays of arrays, in the order specified for the language. In Pascal, `array[a..b,c..d] of T` is equivalent to `array[a..b] of array[c..d] of T` .

Symbol Table Structures for Array

```
var x: array[1..10] of real;
```




```
var z: array[1..5, 1..10] of real;
```



Type Checking, Coercion, and Inference

When a binary operator is reduced, as in our `binop` program, it is necessary to check the types of the arguments, possibly to *coerce* an argument to the correct type, and to infer the result type.

Suppose that `X` is `real` and `I` is `integer`.

Op	Arg1	Arg2	Op	New Arg1	New Arg2	Result
+	X	X	+	X	X	real
+	I	I	+	I	I	integer
+	I	X	+	(float I)	X	real
:=	I	X	:=	I	(fix X)	integer
<	I	X	<	(float I)	X	boolean

- For most operators, the "higher" type takes precedence. Numeric types may be organized in a *type lattice* where e.g. `char < int < float < double`.
- For assignment, the type of the left-hand side takes precedence.
- The result type is generally the operand type, but is always `boolean` for relational operators.

The result type is stored in the operator token.

Structure References

References to parts of structures within program code are not hard to handle. The basic principles are:

- Every expression has a type.
- Types form a tree structure (a graph when pointers to types are included, but still treated as a tree).
- The structure references in source code specify a traversal down the type tree.
- A reference to a part of a structure depends on the type of the structure; the results of the reference are:
  - An address expression for the substructure.
  - A new type for the substructure.

$(code, type) \rightarrow (code', type')$

Repeated application of this procedure handles even complicated references.

There are several basic cases: arrays, records, and pointers.

Structure References....

1. **Arrays:** The address is the base address of the array plus the offset given by the array formula.

   `(AREF` *base offset* `)`

   The type is the array element type.

2. **Records:** The address is the base address of the record plus the offset of the field within the record: `(AREF` *base offset* `)`
   The type is the field type.

3. **Pointers:** The address is given by accessing the *value* of the pointer: `(^` *pointer* `)`
   The type is the type pointed to. This is called *dereferencing* the pointer.

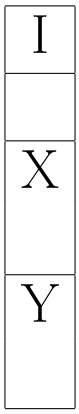
Because structure references generate many additive constants, it is an important and easy optimization to keep additive constants at the front of the addressing expression and to combine (fold) additive constants as structure accesses are processed:

$$Address = BaseAddress + \sum_{i=1}^{n} Offset_i$$

Record References

A record is a data area consisting of a sequence of items, which may be of different sizes and types. Items are addressed by name.

```
TYPE Z = RECORD I: INTEGER; X, Y: REAL;
VAR W: Z;
```



Field:	Offset:	Size:
I	0	4
X	8	8
Y	16	8

`TOTAL SIZE = 24`

`W.X` would produce the code `(AREF W 8)`.

Array References

An array is a data area consisting of a linear sequence of items, each of which is of the same size and type. Items are addressed by numeric index.

The address of an array reference must be calculated by the compiler. We assume that the array is declared as:
`A: ARRAY[` $L_1..U_1, L_2..U_2,\ \ldots\ , L_n..U_n$ `] OF` $Type$

The address of a reference `A[` $S_1, S_2, ..., S_n$ `]` is given by the formula:

$$Address = Base(A) + \sum_{i=1}^{n} (S_i - L_i) \cdot D_i$$

where for row-major order (C, Pascal),

$$D_n = length\ of\ one\ datum\ of\ Type$$

$$D_{i-1} = D_i \cdot (U_i - L_i + 1)$$

or for column-major order (Fortran),

$$D_1 = length\ of\ one\ datum\ of\ Type$$

$$D_{i+1} = D_i \cdot (U_i - L_i + 1)$$

Array References in Pascal

Pascal uses *row-major order*, making array declarations and references easy to process.

For array declarations,

`array[i..j,k..l] of` *type*

is equivalent to:

`array[i..j] of array[k..l] of` *type*

These declarations are reduced starting at the *right*, so that each reduction is of the form:

`array[...] of` *type*

where *type* has been completely processed and has a known size.

For array references,

`a[i,j]`

is equivalent to:

`a[i][j]`

These references are processed left-to-right, so that `a[i]` is done first, yielding code for `a[i]` and a type that is an array type; then `[j]` is processed relative to that array type.

Using these conventions, arrays with multiple dimensions are handled in the same way as singly-dimensioned arrays.

Does Array Order Matter?

```
static double arr[1000][1000];

double test1 ()
{ double sum; int i, j;
  sum = 0.0;
  for (i=0; i<1000; i++)
    for (j=0; j<1000; j++)
      sum = sum + arr[i][j];
  return (sum); }

double test2 ()
{ double sum; int i, j;
  sum = 0.0;
    for (j=0; j<1000; j++)
  for (i=0; i<1000; i++)
      sum = sum + arr[i][j];
  return (sum); }
```

The two programs compute the same result, but performance is quite different:

```
test1 result = 1000000.0    time =  430000
test2 result = 1000000.0    time = 1940000
```

Example of Structure Declaration

```
type date   = record   mo : 1..12;
                      day : 1..31;
                     year : integer end;

     person = record name : alfa;
                       ss : integer;
                    birth : date end;

var people : array[1..100] of person;
```

From these declarations, the following symbol table information might be derived:

Symbol	Offset	Size	Type
`date`	0	16	Link to fields
`mo`	0	4	`1..12`
`day`	4	4	`1..31`
`year`	8	4	`integer`
`person`	0	32	Link to fields
`name`	0	8	`alfa`
`ss`	8	4	`integer`
`birth`	16	16	`date`
`people`	0	3200	`person`

Example of Structure Reference

Given the preceding declarations, the reference

```
people[i].birth.day
```

would be processed as follows:

Reference		Expression	Type
`people[i]`		`aref(people, (i-1)*32)`	`person`
	=	`aref(people, -32 + i*32)`	
`.birth`		`aref(people, (i-1)*32+16)`	`date`
	=	`aref(people, -16 + i*32)`	
`.day`		`aref(people, (i-1)*32+16+4)`	`1..31`
	=	`aref(people, -12 + i*32)`	

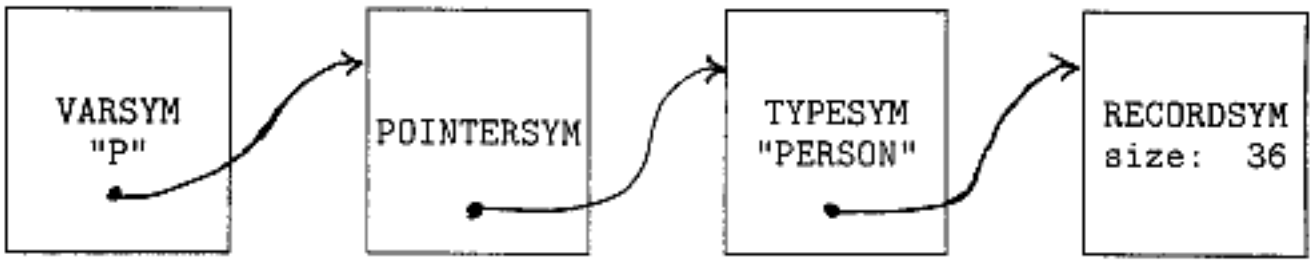
It is useful to think of structure descriptions in nested Lisp-like form. For example, the preceding declarations would yield the following type for the array `people`:

```
(ARRAY (.. 1 100)
       (RECORD (NAME ALFA)
               (SS INTEGER)
               (BIRTH (RECORD
                        (MO (.. 1 12))
                        (DAY (.. 1 31))
                        (YEAR INTEGER)))))
```

Pointer Reference

An example of a pointer declaration is:

```
var p: ^person
```



Note that the `POINTERSYM` record points to the *name* of the record rather than to the `RECORDSYM` entry.

A pointer reference such as:

```
p^.friend
```

is handled like an array reference, with the pointer reference in the base address location:

```
(aref (^ p) 4)
```

where we assume that `4` is the offset of the `friend` field from the start of the `person` record.

The file `pasrec.sample` contains examples of code that is compiled for pointer references.

Types as Tree Structures

Types can be thought of as tree structures:

- Leaf nodes are basic types (integer, real).
- Interior nodes are type constructors (array, record, pointer).

A type declaration builds an interior node from leaf nodes or subtrees.

A structure reference specifies a path through the tree from root to leaf node.

Pointers provide "back arcs" in the tree.

Dynamic Type Checking

Dynamic type checking, used in Lisp, checks types at runtime before operations are performed on data.

- Types are associated with *values*. Requires the ability to determine the type of any data at runtime, e.g. by storing a type *tag* with data. Or, pages of memory can be allocated to specific types; the type of data can be looked up in a table indexed by page address.
- Slower at runtime due to repeated type checking.
- More flexible: union types can be supported, as in the `copy-tree` function.

```
(defun copy-tree (x)
  (if (consp x)
      (cons (copy-tree (car x))
            (copy-tree (cdr x)))
      x))
```

Common Lisp combines dynamic type checking with optional type declarations to allow compilation of code using static checks.

Static Type Checking

Static type associates type with *variables*; checking is done once, at compile time.

- The compiler must be able to determine "all" types at compile time: type declarations required.
- Faster at runtime: no type checks, only operations.
- Inflexible. Pascal originally included array bounds as part of the type, making it impossible to write general subroutines such as matrix inversion.

Strong Typing

A *sound* type system guarantees that no type errors can occur at runtime. A language with such a system is called *strongly typed*. However,

- Some checks can only be done dynamically, e.g. array bounds checking.
- Such checking can cause serious performance degradation (factor of 2).
- Even so-called strongly typed languages often have "holes" in the type system, e.g. variant records in Pascal.

Type Equivalence

If types are required to match, the question of what types are considered to be "equivalent" arises.

Structural equivalence requires equivalent structures (tree equivalence, as in the Lisp function `equal`), i.e.,

- Identical basic types, or
- Same type constructor applied to equivalent types.

However, there is still a question as to what things are included when testing equivalence, e.g., array bounds and record field names. Implementations of the "same" language may vary.

Name equivalence considers types to be equal only if the same names are used.

C uses structural equivalence except for records, for which name equivalence is used.

Type Signatures

The *signature* of a procedure is a description of the types of its arguments (in order) and the type of its result. Often a signature is written in Cartesian product form:

`sqrt` : $real \rightarrow real$
`mod` : $integer \times integer \rightarrow integer$

Procedure signatures must be put in the symbol table when compiling languages that require type-checking of procedure arguments.

In languages such as Ada, which allow separate compilation of procedures but require argument type checking, type signatures must be part of the output file so that type checking of procedure calls can be performed by the link editor.

Polymorphic Procedures

A procedure that has multiple type signatures is called *polymorphic*. In the case of operators, the use of the same operator with different types is called *overloading*.

`+` : $integer \times integer \rightarrow integer$
`+` : $real \times real \rightarrow real$

Polymorphic procedures are often found in object-oriented programming:

```
(defmethod move-to ((self drawable)
                    (x integer) (y integer))
  (send self 'erase)
  (send self 'set-pos x y)
  (send self 'draw) )
```

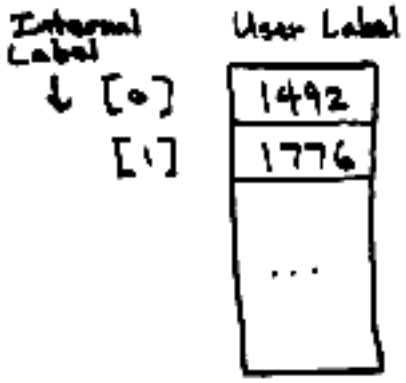
Table for Labels

Pascal requires a `label` statement to pre-declare statement labels:

```
label 1492, 1776;
```

It is necessary to ensure that user labels do not conflict with compiler-generated labels. An easy way to do that is to keep user labels in a linear table, whose index becomes the internal label.

Labels used in statements are converted to the internal labels. The intermediate code has both `label` and `goto` as operators, each taking an integer internal label number as argument.




```
labels[labelnumber++] = tok->intval;
```

```
1776:   (label 1)        goto 1776;   (goto 1)
```

Intermediate Code

Intermediate code is:

- the output of the Parser and the input to the Code Generator.
- relatively machine-independent: allows the compiler to be *retargeted*.
- relatively easy to manipulate (optimize).

Kinds of Intermediate Code

1. Trees
2. Quadruples
3. Triples
4. Polish
5. Assembly Language
6. Machine Language

Trees

Trees are not commonly used as intermediate code, but in my opinion trees are the best:

- Trees are easy to manipulate, optimize, partially evaluate.
- Trees can be transformed using patterns.
- Trees are easily converted to other forms of intermediate code.

The only disadvantage of trees is that they may take more storage; but modern computers have plenty of storage for compilation.

Quadruples

A *quadruple* is a "mini assignment statement" with at most one operator. `X := A + B * C` can be represented as:

```
T1 := B * C
T2 := A + T1
X := T2
```

T1	B	*	C
T2	A	+	T1
X	T2		

The four components of a quadruple are:

1. Result
2. Operand 1
3. Operation
4. Operand 2

This is sometimes called *three-address code* since each quadruple involves three variables.

Advantages:

1. Flat: can be written to a linear file.
2. Easy to generate (rather poor) code.
3. Can perform peephole optimization.

Triples

A *triple* or *triad* is like a quadruple in which the result component is implicitly given by the triple number. `X := A + B * C` might be represented as:

```
(0)  B  *   C
(1)  A  +  (0)
(2)  X  := (1)
```

The three components of a triple are:

1. Operand 1
2. Operation
3. Operand 2

Reverse Polish Notation

Reverse Polish Notation (RPN) is an unambiguous, parenthesis-free notation for arithmetic expressions. [32] Operands precede the operators that apply to them:

```
A + (B * C)    -->  A B C * +

(A + B) * C    -->  A B + C *
```

It is easy to produce RPN as output of an operator precedence parser:

1. Operands go directly to the output; no operand stack is used.
2. When an operator is reduced, it is removed from the operator stack and put into the output.

[32] The description "Polish" refers to the nationality of logician Jan Łukasiewicz, who invented (prefix) Polish notation in the 1920s. – Wikipedia

Trees and Reverse Polish

It is easy to convert a tree representation to RPN by a postorder recursive algorithm:

1. If the expression to be converted is an operand (leaf node), put it into the output.
2. Otherwise, the expression is a subtree. Convert the left-hand operand, then convert the right-hand operand, then put the operator into the output.

Converting a Tree to RPN

```
(defun tree-to-polish (exp)
  (nreverse (tree-to-p exp nil)))

(defun tree-to-p (exp result)
  (if (atom exp)
      (cons exp result)
      (progn
        (mapc #'(lambda (opnd)
                  (setq result
                        (tree-to-p opnd
                                   result)))
              (rest exp))
        (cons (first exp) result))))

(tree-to-polish '(+ (* a b) c))  =  (A B * C +)

(setq testx '(/ (+ (minus b)
                   (sqrt (- (expt b 2)
                            (* 4 (* a c)))))
                (* 2 a)))

(tree-to-polish testx)
 = (B MINUS B 2 EXPT 4 A C * * - SQRT + 2 A * /)
```

Executing Reverse Polish

RPN is easily executed by a stack machine:

1. Operands are pushed onto the stack.
2. When an operator is encountered, the appropriate number of operands is popped from the stack and the operator is applied to them. The result is pushed onto the stack.
3. When the end of the input expression is reached, the result is the top of the stack.

HP makes a line of pocket calculators that use RPN.

Executing RPN

```
(defun execute-rpn (rpn)
  (let (stack rhs)
    (dolist (item rpn)
      (if (numberp item)
          (push item stack)
          (progn (setq rhs (pop stack))
                 (push (if (unaryp item)
                           (funcall item rhs)
                           (funcall item
                                    (pop stack)
                                    rhs))
                       stack))))
    (pop stack) ))

(defun unaryp (x) (member x '(minus sqrt)))

(execute-rpn '(3 4 + 5 *))  =  35

(execute-rpn (sublis '((a . 1)
                       (b . -3)
                       (c . 2))
                     (tree-to-polish testx)))
  =  2.0
```

RPN as Intermediate Code

An entire program can be expressed in Reverse Polish intermediate code. For example, by reading the pretty-printed intermediate code from `graph1.pas` into Lisp and converting it to RPN, the following is obtained:

```
(I 0 := 1 LABEL I 32 <= X 0.0625 I FLOAT * :=
 Y EXP X - FUNCALL SIN 6.28318 X * FUNCALL * :=
 N ROUND 32.0 Y * FUNCALL 34 + := 0 LABEL
 WRITE " " FUNCALL N N 1 - := N 0 = PROGN
 0 GOTO IF PROGN WRITELN "*" FUNCALL
 I I 1 + := 1 GOTO PROGN IF PROGN)
```

It must be clear how many operands each operator takes, so a construct such as `progn` must be replaced by `prog2`, `prog3`, etc.

Code Generation

The Code Generator is the final stage of the compiler. It accepts intermediate code from the Parser and produces machine code for a particular target machine.

Output code may be:

- *Absolute code* with fixed memory addresses, either to a file or directly into memory.
- *Relocatable code* that is converted to absolute code by a *link editor*.

Loading Process

An absolute file is loaded and executed by the *loader* (part of the operating system) in the following way:

1. Allocate memory for each program segment.
2. Read the memory image into main memory from the absolute file on disk.
3. Set the *base register* to the initial address of the segment and set the *bounds register* to the size of the segment.
4. Jump to the starting address of the program, using a special jump instruction that switches the processor state from *system* mode to *user* mode during the jump.

Absolute File

An *absolute file* is a file of program code and data that is ready for loading and execution. It is *absolute* in the sense that every address is an absolute (numeric) address.

An absolute file consists of:

- A `size` and `starting address`
- A memory image for the code segment, and perhaps a memory image for the data segment.
- A list of `BSS` (Block Starting with Symbol) elements:
  - Starting address
  - Size
  - Initial data value

Initializing BSS Storage

A block of data can be declared in assembly language with a `BSS` (Block Starting with Symbol) pseudo-op:

```
BIGARRAY:     BSS         8000000
```

Similar data can be declared in high-level languages:

```
DOUBLE PRECISION BIGARRAY(1000,1000)
```

Instead of representing such a large block as a memory image, the block is represented in the absolute file by giving its starting address, size, and the initial value. The loader will fill the memory area with the initial value.

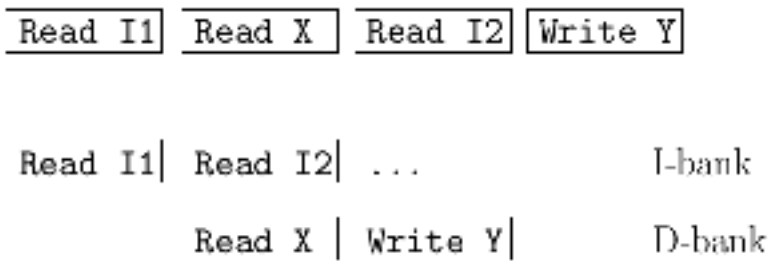
Although the loader might not have to initialize `BSS` areas, there are several advantages to doing so:

- It prevents security problems: otherwise, a program can see the previous program's data.
- It makes a program run the same way each time. Otherwise, a program might work if the initial value is "lucky", but not work for other values.
- It allows detection of use of uninitialized data if the array is filled with the SNaN (Signaling Not a Number) value.

Banked Memory

Some CPU architectures employ a *banked memory* in which memory is divided into two *banks* that can be accessed independently. This allows instruction fetching (from the instruction bank) to be overlapped with data access (from the data bank).

```
I1:   LDA  X
I2:   STA  Y
```



With banked memory, the CPU can run twice as fast. On such a machine, the compiler and link editor will have to put code and data in separate banks.

Location Counter

As code is generated, it will be put into the next available location in a static data area. A *location counter* keeps track of the next available location in the data area.

1. Initially, the `LC` is set to `0` .
2. After each instruction is generated, the `LC` is incremented by the length of the instruction.
3. When a label is encountered, the relative address of the label is set to the value of the `LC`.
4. At the end, the length of the code segment is the value of the `LC`.

There may be multiple location counters, *e.g.*, one for code and one for static data.

Example of Assembly Listing

```
 1    0000                          #9
 2    0000                                  global  _GRAPH1
 3    0000                          _GRAPH1:
 4    0000    480E FFFF FFE8                link.l  %a6,&LF1
 5    0006    48D7 0000                     movm.l  &LS1,(%sp)
 6    000A    F236 F000 0170 FFFF           fmovm.x &LSF1,LFF1(%a6)
              FFE8
 7    0014    7000                          movq.l  &0,%d0
 8    0016    2D40 FFF8                     mov.l   %d0,-8(%a6)
 9    001A                          L1:
10    001A    202E FFF8                     mov.l   -8(%a6),%d0
11    001E    7220                          movq.l  &32,%d1
12    0020    B280                          cmp.l   %d1,%d0
13    0022    6DFF 0000 00A2                blt.l   L3
14    0028    F239 5400 0000 00CC           fmov.d  L4,%fp0
15    0030    F200 4080                     fmov.l  %d0,%fp1
16    0034    F200 0423                     fmul    %fp1,%fp0
17    0038    F22E 7400 FFE8                fmov.d  %fp0,-24(%a6)
18    003E    F200 001A                     fneg    %fp0
19    0042    F200 0010                     fetox   %fp0
20    0046    F239 5480 0000 00D4           fmov.d  L5,%fp1
21    004E    F22E 5500 FFE8                fmov.d  -24(%a6),%fp2
22    0054    F200 08A3                     fmul    %fp2,%fp1
      ...
50    00BC    2D40 FFF8                     mov.l   %d0,-8(%a6)
51    00C0    60FF FFFF FF58                bra.l   L1
52    00C6                          L3:
53    00C6    4E5E                          unlk    %a6
54    00C8    4E75                          rts
55    00CA                                  set     LF1,-24
56    00CA                                  set     LS1,0
57    00CA                                  set     LFF1,-24
58    00CA                                  set     LSF1,0
59    00CC                                  data
60    00CC                                  lalign  4
61    00CC    3FB0 0000 0000 0000   L4:     long    0x3fb00000,0x0 # 6.25e-02
62    00D4    4019 21FA 0000 0000   L5:     long    0x401921fa,0x0
63    00DC    4040 0000 0000 0000   L6:     long    0x40400000,0x0
64    00E4    2000                  L7:     byte    32,0
65    00E6    2A00                  L8:     byte    42,0
66    00E8                                   version 2
```

Backpatching

In order to compile code for branch instructions, it is necessary to know the address in memory (relative to the location counter) of the destination of the branch. Branch instructions may be PC-relative or have absolute addresses. For *forward branches* the destination address will not be known when the branch is encountered. There are two approaches to this problem:

1. Two passes. The first pass through the code generator computes addresses but does not generate code. The second pass generates code and uses the addresses determined in the first pass. This approach takes longer.
2. Backpatching. The generated code itself is used to hold a linked list of places where forward reference addresses are needed. When the address is found, it is patched back into the generated code.

Link Editing Process

The task of the *link editor* is to convert a set of relocatable files into an absolute file that can be executed. The link editor is called `ld` in Unix; it is called automatically by compilers in most cases.

Link editing involves several steps:

1. Determine the relocatable code modules that are needed.
2. Allocate memory and assign absolute addresses for each module.
3. Determine absolute addresses of external symbols.
4. Relocate the code to produce an absolute code file.

Relocatable Code

A relocatable code file contains the following items:

1. Size of each static data area; each data area corresponds to a *location counter*. There may be one data area, two for code and for data, or many.

LC	Size
0	200
1	400

2. Table of *imported symbols*: external symbols (not defined in that module) that are referenced by the code.

N	Name
0	SIN
1	EXP

3. Table of *exported symbols* or entry points; for each, its location counter number and offset from that location counter.

Name	LC	Offset
GRAPH1	0	40

4. Data or code for each data area.

Finding Relocatable Modules

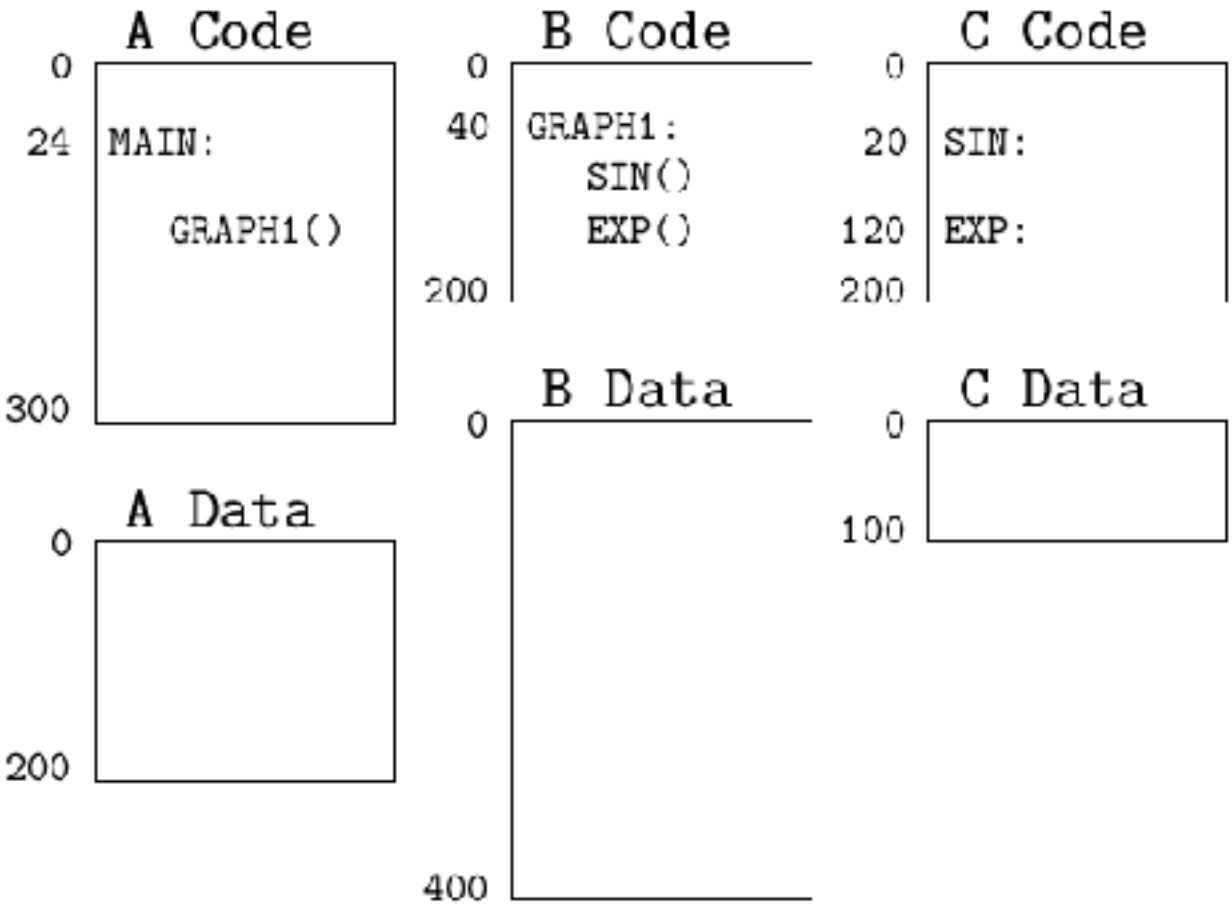
The link editor is given the name of one or more program modules that should be included. These form the initial module set.

1. For each module, the exported symbols and imported symbols are collected.
2. If there is some imported symbol that it not defined, it is necessary to find a relocatable module that defines it. This is done by searching through one or more *libraries*, which are collections of relocatable files. This process typically occurs for system library routines such as `sin` or `writeln`.
3. The above processes are repeated until no undefined symbols remain. If a symbol cannot be found, an *unsatisfied external reference* error is reported.
4. If any external symbol is defined more than once, a *multiply defined external symbol* error is reported.

Assigning Absolute Addresses

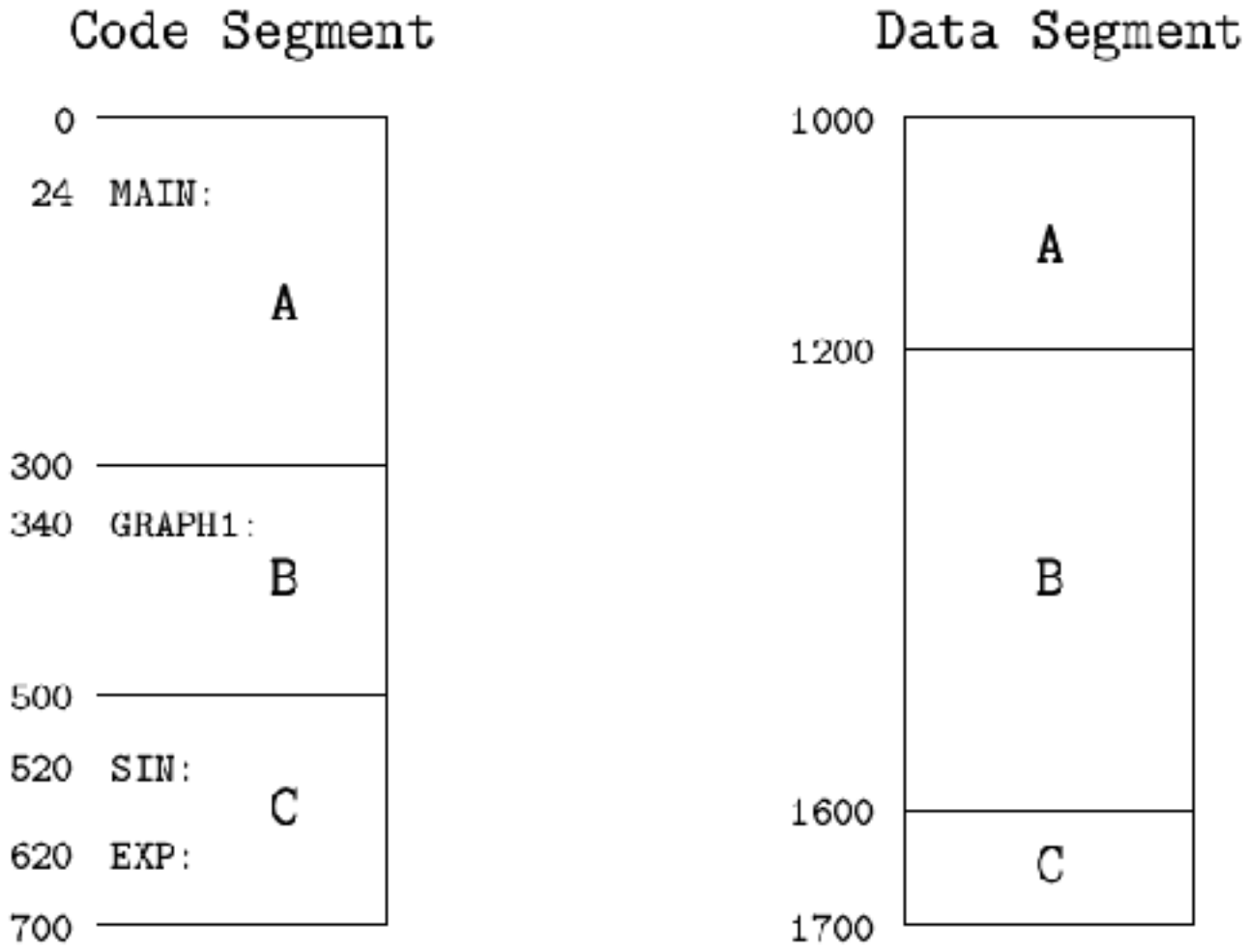
Absolute addresses are assigned using the storage allocation algorithm for each module. If the CPU has a banked memory, the instruction and data segments will be allocated in separate banks.

Suppose that the set of relocatable modules to be loaded is `A`, `B`, and `C`, with sizes and addresses as shown below:



Absolute Addresses

Once absolute addresses have been assigned to each storage block (location counter) of each module, the absolute address of each entry point can be determined.



Link Editor

At this point, the link editor has constructed a set of tables:

1. An absolute memory address for each data area. This address is used to relocate references to data in this area.

File	LC	Address
A	0	0000
A	1	1000
B	0	0300
B	1	1200
C	0	0500
C	1	1600

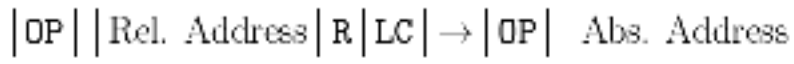
2. External symbols and their addresses:

Name	Address
MAIN	0024
GRAPH1	0340
SIN	0520
EXP	0620

The next step is to convert the code and data to absolute form.

Form of Relocatable Code

Program code consists of instructions; each instruction may have an address field that contains the *offset* of a variable or label from its data area. In addition, the instruction will have a relocation bit or field that indicates whether the address should be relocated. If the relocation bit is set, the corresponding location counter is added to the address.



`R` is a relocation code; it could be:

1. Do not modify the word.
2. Relocate the address field by adding the value of location counter `LC` to it.
3. Replace the address field by the address of the external symbol whose number is given in the address field.

Relocatable code may contain other kinds of code items, *e.g.*, `BSS` codes to specify blocks to be filled with a specified value such as `0`.

Static Allocation

For a static language, such as Fortran, all program code and data are allocated at fixed addresses in memory.

`COMMON` storage is data storage that is accessible from multiple programs. The name of a `COMMON` block is an external symbol that names a set of data that is accessible from multiple programs.

```
COMMON /ARRAYS/ X(1000), Y(1000)
```

Dynamically Linked Library

A *dynamically linked library* (DLL) is not part of an absolute file, but is linked in at load time or runtime. The primary motivation of a DLL is to allow multiple programs to share a single copy of a large library, such as X Windows. DLLs also allow a library update (e.g. security patch) to be effective for all programs without re-linking.

Ways to link in the DLL include:

- Let the loader do the linking at load time. The absolute file contains information on what external symbols are needed and where they go.
- Let the program make calls to initialize itself on first call.
- Leave the calls undefined, so that they generate error traps at runtime. The OS figures out what the calls should be and fixes the code at the point of call.

Run-Time Support

Run-time support includes:

1. Library routines (math functions, I/O formatting)
2. Interface with operating system (I/O, termination)
3. Memory management (`new`, garbage collection)
4. Debugging support
5. Procedure call support (often compiled as part of a procedure)

Operations by Subroutine

Functions may be specified in source code either as operators or as function calls; the compiler may generate either form as instructions or as a subroutine call.

A function can be compiled *inline* if the code is short:

```
abs(x)   -->  (if (>= x 0) x (- x))
```

or the code generator can generate a special instruction sequence, e.g. to float an integer.

An operator can be compiled as a *subroutine call* if the code is longer.

```
X**Y   -->   qq8rtor(x,y)
```

```
cout << "Hello\n"
```

On small machines without floating point hardware, all floating ops may be done by subroutine calls.

Special Subroutines

In C, `printf` is an ordinary subroutine.

However, in Pascal `write` *cannot* be just an ordinary subroutine because `write` must know the types of its arguments in order to convert them correctly. This can be done in two ways:

- The compiler can add a type argument to the call to the library routine:
  `write(x)` → `write(x, REAL)`
- The compiler can call different library routines depending on the argument type:
  `write(x)` → `writef(x)`

We will assume the second convention for our assignments; we will use `write` and `writeln` for strings, `writei` and `writelni` for integers, and `writef` and `writelnf` for floats.

Memory Management

Many languages can allocate *heap* storage at runtime.

```
type person = record ... end;
var p: ^person;
  ...
  new(p);
```

The compiler must insert the size of the `person` record into the call to `new`. If type safety is to be guaranteed, it may be necessary to initialize the new record.

For our assignments, we will assume the following conversion:

`new(p)` $\rightarrow$ `p := new(sizeof(p^))`

For example, source code `new(p)` would be converted to intermediate code `(:= p (funcall new 40))`, where `40` is the size of the record type `p` points to.

Pascal uses the form `new(p)` and prohibits expressions involving pointers in order to prohibit pointer arithmetic (which destroys type safety). C allows pointer arithmetic; in C, `malloc` is called with the record size as a parameter, and a type cast is used to set the result type.

Returning Memory

Some languages allow requested storage to be freed or returned to the heap. This is an efficient method of storage management, but it allows two kinds of errors:

- Forgetting to return memory: a *storage leak*.
- Returning the same memory more than once. This can produce errors that are very difficult to find.

Released storage can be held in linked lists that are pointed to by an array of pointers, where the array is indexed by block size.

Potential problem: mismatch of block sizes. The user requests and releases many blocks of size 7, then needs blocks of size 5.

Larger blocks can be subdivided, but this leads to *fragmentation* (many small blocks that cannot be used). The *buddy system* (see Knuth) is one way of combining smaller blocks into larger ones when possible.

`malloc` puts the size of a block *behind* the block (just before the pointer value that is returned). This makes it easy to check whether adjacent blocks have been returned, so that they can be combined. However, a misbehaving program can overwrite the size and crash the system.

Heap Memory Management

```
void * new(size)
  long size;
  { CONS * tmp;
    if ( free[size] )
      { tmp = (CONS *) free[size];
        free[size] = tmp->car;
        return (tmp); }
    else if ( (void *) ( (int) nextfree + size)
                                    < maxfree )
      { tmp = nextfree;
        nextfree = (void *)
                   ( (int) nextfree + size);
        return (tmp); }
    else error("Out of Memory");
  }

release(block, size)
  void * block; long size;
  {  ((CONS *) block)->car = free[size];
     free[size] = block;
   }
```

Garbage Collection

Garbage collection is a method of automatically recycling storage that is no longer in use:

- If heap storage is available, return the next item of heap storage.
- Otherwise, perform a garbage collection to reclaim unused storage. If enough was collected, allocate the requested item and continue.
- Otherwise, request more memory from the operating system.
- Otherwise, fail due to lack of memory.

Garbage collection requires that the type of every piece of runtime memory be identifiable by the garbage collector, and that there are no possible errors in type determination. This may difficult for some languages, especially if the language allows variant records or pointer arithmetic. Garbage collection has been used in Lisp since about 1960; it is also used in Java.

Garbage Collection

Garbage collection identifies the cells of memory that are in use; the remaining cells, which are not used for anything, are collected and added to the Free List. This automatic recycling of unused memory is a major advantage of Lisp; it makes it possible for programs to create ("`cons` up") new structures at will without having to worry about explicitly returning unused storage.

Identification of "in use" memory starts from symbols, which are always "in use". Symbols, in turn, may have several pointers to other data structures:

1. Binding (value) of the symbol.
2. Function Definition.
3. Property List.

Each of these structures, if present, must also be marked as being "in use".

Mark-And-Sweep Garbage Collection

Mark-and-sweep garbage collection first marks all storage cells that are in use, then sweeps up all unmarked cells. Symbol cells are marked, and all pointers from the symbols are followed using the following recursive algorithm:

1. If the pointer points to a Symbol or to a marked cell, do nothing.
2. Otherwise (pointer points to a `cons` Cell),
   (a) Mark the cell itself.
   (b) Apply the marking algorithm to the `car` of the cell.
   (c) Apply the marking algorithm to the `cdr` of the cell.

Mark-and-Sweep ...

After all cells that are in use have been marked, the Sweep phase is run. All memory cells are examined, in order of increasing address. Those cells that are not marked are pushed onto the Free List.

"Marking" a cell may use an available bit within the cell, or it may use a separate bit table that uses one bit to represent each word.

Mark-and-Sweep garbage collection is conceptually simple. However, it requires time that is proportional to the total size of the *address space*, independent of how much garbage is collected. This is a disadvantage for large address spaces.

Another disadvantage of this algorithm is that all computation stops for several seconds while garbage is collected. This is not good for real-time applications, e.g., a robot walking down stairs.

Copying Garbage Collection

Another method of garbage collection is to divide the total address space of the machine into two halves. When storage is exhausted in one half, garbage collection occurs by copying all storage that is in use to the other half. Unused storage, by definition, doesn't get copied.

A copying collector uses time proportional to the amount of storage that is *in use*, rather than proportional to the address space. This is advantageous for programs that generate lots of garbage. Copying also tends to put list structures in nearby addresses, improving memory locality.

A copying collector has two disadvantages:

1. Half the address space of the machine may be lost to Lisp use, depending on the implementation.
2. There is a long period during which computation stops for garbage collection.

Reference Counting

Another method of managing Lisp storage involves *reference counting*. Conceptually, within each `cons` cell there is room for a small counter that counts the number of pointers which point to that cell. Each time another pointer to the cell is constructed, the counter is incremented by one. Each time a pointer is moved away from the cell, the counter is decremented by one.

Whenever the reference count of a cell becomes zero, the cell is garbage and may be added to the Free List. In addition, the reference counts of whatever *its* pointers point to must also be decremented, possibly resulting in additional garbage.

Reference Counting...

Advantages:

1. Garbage collection can be *incremental*, rather than being done all at once. Garbage collection can occur in short pauses at frequent time intervals, rather than in one long pause.
2. Time spent in collection is proportional to *amount collected* rather than to *address space*.

Disadvantages:

1. More complexity in system functions (`cons`, `setq`, `rplaca`, etc.).
2. Requires storage bits within each `cons` cell, or other clever ways of representing counts.
3. Cannot garbage-collect circular structures (since reference count never becomes zero).

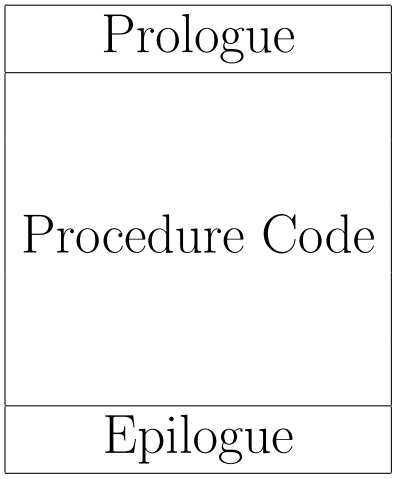
Garbage Collection Is Expensive

Garbage collection is safer and more convenient for the programmer than explicit releasing of storage.

However, whatever the implementation, garbage collection is usually computationally expensive. As a rough rule of thumb, one can think of a `cons` as taking 100 times as much CPU time as a basic instruction.

The moral: avoid unnecessary `cons`es.

Compiled Procedure



Prologue: (or *preamble*) Save registers and return address; transfer parameters.

Epilogue: (or *postamble*) Restore registers; transfer returned value; return.

A `return` statement in a procedure is compiled to:

1. Load the returned value into a register.
2. `goto` the Epilogue.

Subroutine Call Is Expensive

The prologue and epilogue associated with each procedure are "overhead" that is necessary but does not do user computation.

- Even in scientific Fortran, procedure call overhead may account for 20% of execution time.
- Fancier languages have higher procedure call overhead.
- Relative overhead is higher for small procedures.
- Breaking a program into many small procedures increases execution time.
- A `GOTO` is much faster than a procedure call.
- Modern hardware architecture can help:
  - Parameter transfer
  - Stack addressing
  - Register file pointer moved with subroutine call

Activations and Control Stack

An *activation* is one execution of a procedure; its *lifetime* is the period during which the procedure is active, including time spent in its subroutines.

In a recursive language, information about procedure activations is kept on a *control stack*. An *activation record* or *stack frame* corresponds to each activation.

The sequence of procedure calls during execution of a program can be thought of as a tree. The execution of the program is the traversal of this tree, with the control stack holding information about the active branches from the currently executing procedure up to the root.

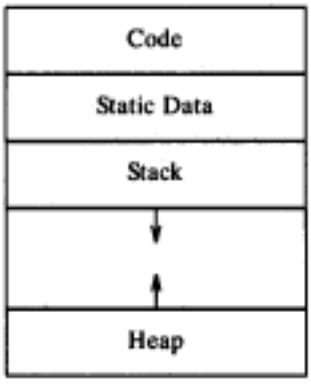
Environment

The *environment* of a procedure is the complete set of variables it can access; the *state* of the procedure is the set of values of these variables.

A *binding* is an association of a name with a storage location; we use the verb *bind* for the creation of a binding and say a variable is *bound* to a location. An environment provides a set of bindings for all variables.

An assignment, e.g. `pi := 3.14` , changes the state of a procedure but not its environment.

Run-time Memory Organization



[Aho, Sethi, and Ullman, *Compilers*, Fig. 7.7.]

Code Generation

We assume that the input is error-free and complete, for example that any type conversion operators have already been inserted.[33]

Can generate:

- Binary
  - absolute
  - relocatable
- Assembly
- Interpreted code (e.g. Java byte codes)

Problems include:

- Instruction selection
- Register management
- Local optimization

[33] This slide was written by John Werth.

Code Generation

Code generation can be broken into several steps:

1. Generate the prologue
2. Generate the program code
3. Generate the epilogue

Subroutines are provided to generate the prologue and epilogue.

The arguments to the code generator are:

```
gencode(pcode, varsize, maxlabel)

   pcode    = pointer to code:
              (program foo (progn output)
                           (progn ...))
   varsize  = size of local storage in bytes
   maxlabel = max label number used so far
```

Code Generation

A starter program `codgen.c` is furnished. A very simple program, `triv.pas`, can be compiled by `codgen.c`:

```
program graph1(output);
var i:integer;
begin   i := 3   end.
```

The result is `triv.s`:

```
.globl graph1
   .type graph1, @function
graph1:
   ...
   subq $32, %rsp      # space for stack frame
# --------- begin Your code -------
   movl $3,%eax        #  3 -> %eax
   movl %eax,-32(%rbp) #  i := %eax
# --------- begin Epilogue code ---
   leave
   ret
```

Running Generated Code

Programs can be run using `driver.c` as the runtime library:

```
% cc driver.c triv.s -lm
% a.out
calling graph1
exit from graph1
```

`driver.c` is quite simple:

```
void main()
  { printf("calling graph1\n");
    graph1();
    printf("exit from graph1\n");
  }

void write(char str[])
  {  printf("%s", str);  }

void writeln(char str[])
  {  printf("%s\n", str);  }

int round(double x)
...
```

Overview of Code Generation

We will take a hierarchical approach to code generation:

- `genc(code)` generates code for a statement. There are only a few kinds of statements. `genc` is easy to do given `genarith`.
- `genarith(expr)` generates code for an arithmetic expression. `genarith` is a classical postorder tree-recursive program, with a simple basic form (but many special cases). `genarith` is not hard given `getreg`.
- `getreg` gets a register from a pool of available registers. It also handles returning unused registers.
- While register management can be complex, a simple implementation works pretty well. We will discuss some improvements.

Code Generation for Statements

The function `genc(code)` generates code for a statement. There are only a few kinds of statements:

1. `PROGN`
   For each argument statement, generate code.
2. `:=`
   Generate the right-hand side into a register using `genarith`. Then store the register into the location specified by the left-hand side.
3. `GOTO`
   Generate a Branch to the label number.
4. `LABEL`
   Generate a Label with the label number.
5. `IF`
   `(IF c p1 p2)` can be compiled as:
   `IF c GOTO L1;`
   `p2; GOTO L2; L1:  p1; L2:`
   Optimizations are discussed later.
6. `FUNCALL`
   Compile short *intrinsic* functions in-line. For others, generate subroutine calls.

Arithmetic Expressions

Code for arithmetic expressions on a multi-register machine can be generated from trees using a simple recursive algorithm.

The specifications of the recursive algorithm are:

- **Input:** an arithmetic expression tree
- **Side Effect:** outputs instructions to the output file
- **Output:** returns the number of a register that contains the result.

Basic Expression Algorithm

The basic algorithm for expressions is easy: postorder.

- Operand (leaf node): get a register; generate a `load`; return the register.
- Operator (interior node): generate operand subtrees; generate op; free operand register; return result register.

```
(defun genarith (x)
  (if (atom x)                ; if leaf,
      (genload x (getreg))    ;   load
      (genop (op x)           ; else op
             (genarith (lhs x))
             (genarith (rhs x))) ) )

>(genarith '(* (+ a b) 3))
   LOAD  A,R1
   LOAD  B,R2
   ADD   R1,R2
   LOAD  3,R3
   MUL   R2,R3
R3
```

Trace of Expression Algorithm

```
>(genarith '(* (+ a b) 3))

  1> (GENARITH (* (+ A B) 3))
    2> (GENARITH (+ A B))
      3> (GENARITH A)
        4> (GENLOAD A R1)
   LOAD  A,R1
        <4 (GENLOAD R1)
      <3 (GENARITH R1)
      3> (GENARITH B)
        4> (GENLOAD B R2)
   LOAD  B,R2
        <4 (GENLOAD R2)
      <3 (GENARITH R2)
      3> (GENOP + R1 R2)
   ADD   R1,R2
      <3 (GENOP R2)
    <2 (GENARITH R2)
    2> (GENARITH 3)
      3> (GENLOAD 3 R3)
   LOAD  3,R3
      <3 (GENLOAD R3)
    <2 (GENARITH R3)
    2> (GENOP * R2 R3)
   MUL   R2,R3
    <2 (GENOP R3)
  <1 (GENARITH R3)
R3
```

Arithmetic Expression Algorithm

The `genarith` input is a tree (operand or operator):

- Operand (leaf node):
  1. Get a register.
  2. An operand may be a variable or constant:
     (a) Variable: Generate an instruction to load the variable into the register.
     (b) Constant:
        i. Small constant: Generate an immediate instruction to load it into the register directly.
        ii. Otherwise, make a *literal* for the value of the constant. Generate an instruction to load the literal into the register.
  3. Return the register number.
- Operator (interior node):
  1. Recursively generate code to put each operand into a register.
  2. Generate the operation on these registers, producing a result in one of the source registers.
  3. Mark the other source register unused.
  4. Return the result register number.

Register Management

Issues are:[34]

- register allocation: which variables will reside in registers?
- register assignment: which specific register will a variable be placed in?

Registers may be:

- general purpose (usually means integer)
- float
- special purpose (condition code, processor state)
- paired in various ways

[34]This slide was written by John Werth.

Simple Register Allocation

Note that there may be several classes of registers, *e.g.*, integer data registers, index registers, floating point registers.

A very simple register allocation algorithm is:

1. At the beginning of a statement, mark all registers as not used.
2. When a register is requested,
   (a) If there is an unused register, mark it used and return the register number.
   (b) Otherwise, punt.

On a machine with 8 or more registers, this algorithm will almost always work. However, we need to handle the case of running out of registers.

Heuristic for Expressions

The likelihood of running out of registers can be reduced by using a heuristic in generating code for expressions:

> Generate code for the *most complicated* operand first.

The "most complicated" operand can be found by determining the size of each subtree. However, simply generating code for a subtree that is an operation before a subtree that is a simple operand is usually sufficient.

With this simple heuristic, on a machine with 8 or more registers, the compiler will never[35] run out.

If a machine allows arithmetic instructions to be used with a full address, the operation may be combined with the last load.

[35] Well, hardly ever.

Improving Register Allocation

The simple register allocation algorithm can be improved in two ways:

- Handle the case of running out of available registers. This can be done by storing some register into a temporary variable in memory.
- Remember what is contained in registers and reuse it when appropriate. This can save some load instructions.

Register Allocation

Used	Use Number	Token

An improved register allocation algorithm, which handles the case of running out of registers, is:

1. At the beginning of a statement, mark all registers as not used; set use number to 0.
2. When an operand is loaded into a register, record a pointer to its token in the register table.
3. When a register is requested,
   (a) If there is an unused register: mark it used, set its use number to the current use number, increment the use number, and return the register number.
   (b) Otherwise, find the register with the smallest use number. Get a temporary data cell. Generate a Store instruction (*spill code*) to save the register contents into the temporary. Change the token to indicate the temporary.

Now, it will be necessary to test whether an operand is a temporary before doing an operation, and if so, to reload it. Note that temporaries must be part of the stack frame.

Example of Code Generation

Expression: (A+B) * (C+3.)

Initially:

NUSE = 0 NTEMP = 0

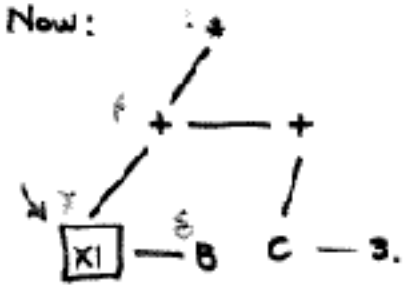
NODE = α

	Pointer	Used	Use No.
X1		0	
X2		0	
X6		0	

1. Operands of α not in registers. Push down to β.
2. Operands of β not in registers. Push down to γ.
3. γ is a variable. Get a register (X1)

   Generate SA1 Bo+A

Now:



NUSE = 1

NODE = γ

	Pointer	Used	Use No.
X1	γ	1	1
X2		0	
X6		0	

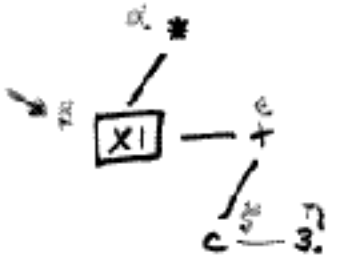
4. Move to next operand (S). Get a register (X2)

   Generate SA2 Bo+B

	Pointer	Used	Use No.
X1	γ	1	1
X2	S	1	2
X6		0	

Example (2)

5. No more operands; pop up to β.
Operands of β are in Registers, so we can generate code.
Assume use of first operand register for Result.



Generate RX1 X1+X2

Release Register X2

Release Nodes γ and δ

Change Node β

	Pointer	Used	Use No.
X1	β	1	3
X2		0	
X8		0	

6. Move across to next element at same level (ε).
Its operands are not in registers, so push down to ζ.

7. ζ is a variable. Get a register (X2).

Generate SAZ B0+C

	Pointer	Used	Use No.
X1	β	1	3
X2	ζ	1	4
X6		0	

8. Move across to element at same level (η).
It is a constant. Look up in constant table; assume its offset in constant table is 4.

Get a Register: No loadable register is free.
Select register with smallest use number (X1).

Example (3)

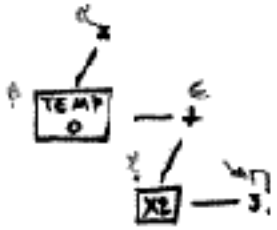
Generate a Store of XL: BX6 X1
SA6 B0+TS+0

(The 0 is the current value of NTEMP)

NTEMP = NTEMP +1

Change node β to a temporary node.

NEWREG = X1



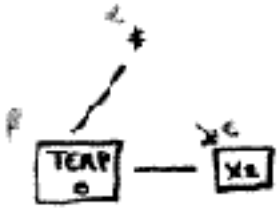
Now we can load the constant 3. :

SA1 B0+CS+4

	Pointer	Used	Use No.
X1	η	1	5
X2	ζ	1	4
X6		0	

9. No more operands, pop up to ε.
Operands of ε are in registers, so we can generate code.
Assume first operand register for result.

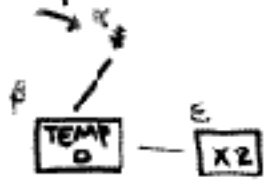
Generate RX2 X2+X1
Release Register X1
Release Nodes ζ and η
Change node ε to Register.



	Pointer	Used	use No.
X1		0	
X2	ε	1	6
X6		0	

Example (4)

10. No more operands at this level. Pop up to Node α.



11. An operand of Node α is not a register. Push Down to β.

12. β is a Temporary. Get a Register (X1)

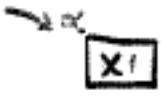
Generate Load: SAI BO+T$+0

	Pointer	Used	Use No.
X1	β	1	7
X2	ε	1	6
X6		0	

13. Move to next node at same level (e). It is a register, so no action is taken.

14. No more nodes at this level, so pop up to Node α.

15. Arguments of Node α are Registers, so we generate code.



Generate RX1 X1 * X2

Release Register X2

Release Nodes β and ϵ

Change node α to Register.

	Pointer	Used	Use No.
X1	α	1	8
X2		0	
X6		0	

16. Top node is a register, so Stop; Result is in X1.

Reusing Register Contents

Used	Contents

Many instructions can be eliminated by reusing variable values that are already in registers:[36]

1. Initially, set the contents of each register to `NULL`.
2. When a simple variable is loaded, set the `contents` of the register to point to its symbol table entry.
3. When a register is requested, if possible choose an unused register that has no contents marked.
4. When a variable is to be loaded, if it is contained in an unused register, just mark the register used. This saves a Load instruction.
5. When a register is changed by an operation, set its contents to `NULL`.
6. When a value is stored into a variable, set the contents of any register whose contents is that variable to `NULL`. Then mark the register from which it was stored as containing that variable.
7. When a Label is encountered, set the contents of all registers to `NULL`.
8. The *condition code* contents can be reused also.

[36] We assume that there are no *aliases* for variables.

Register Targeting

On some machines, it is useful to be able to tell `genarith`, top-down, that its result should be produced in a certain register if possible.

Example: Suppose that a function argument should be transmitted in register `%xmm0`. If the argument can be generated in `%xmm0` directly, it will save a move instruction.

x86 Processor

We will assume an x86-64 processor. This processor has a vast number of instructions (some undocumented) and two major families of assembler syntax and calling sequence conventions. We will use the AT&T/Unix syntax and **gcc** calling conventions.

Integer registers can be used in several sizes, e.g. `%eax` (32 bits) and `%rax` (64 bits) are the same register in different sizes. We will assume that `integer` will use the 32-bit size, while memory addresses must use the 64-bit size.

General-purpose (Integer) Registers:

32/64 bits, numbered `0 - 15` in `genasm`. We will use them in the order `%eax`, `%ecx`, `%edx`, `%ebx` since `%ebx` is callee-saved. `RBASE` to `RMAX` is the local integer register range.

Floating Point Registers:

64 bits, numbered `16 - 31` in `genasm`. `FBASE` to `FMAX` is the floating register range. These are called `%xmm0` through `%xmm7`.

Move (Load/Store) Instructions

Most of the instructions used in a computer program are instructions that move data. The x86 processor uses variable-length instructions and offers very flexible addressing options.

The Unix syntax of x86 instructions shows data movement from left to right:

```
movl    $0,%eax           #  0 -> %eax
movl    %eax,-32(%rbp)    #  %eax -> i
```

There are three data formats that we will use:

Instruction	**Terminology**	**Bits**	**Use For**
`MOVL`	long	32	Integer
`MOVQ`	quad-word	64	Pointer
`MOVSD`	signed double	64	Float

Kinds of Move Addressing

There are several addressing styles that are used with move instructions:

Constants or *immediate* values are specified with a `$`. x86 allows even very large integer constants.

```
movl   $0,%eax          #  0 -> %eax
```

Stack Variables have negative offsets relative to `%rbp`. The offset is the offset from the symbol table minus the stack frame size.

```
movl   %eax,-32(%rbp)   #  %eax -> i
```

In this case, `i` has an offset of 16 and the stack frame size is 48.

Literals have offsets relative to `%rip`.

```
movsd   .LC5(%rip),%xmm0  #  0.0625 -> %xmm0
```

Record References have offsets relative to a register containing a pointer to the record.

```
movl   %eax,32(%rcx)    #  ^. []
```

Move with Calculated Address

x86 allows very flexible addressing:

Offset from Register

```
movl   %eax,-32(%rbp)  #  %eax -> i
```

Offset from Two Registers

```
movsd %xmm0,-1296(%rbp,%rax) #  ac[]
```

The offset and contents of the two registers are added to form the effective address.

Offset from Two Registers with Multiplier

```
movsd %xmm0,-1296(%rbp,%rax,8) #  x[]
```

In this case, the second register is multiplied by 2, 4, or 8 before being added. This can allow many aref expressions to be done in a single instruction.

Literals

A *literal* is constant data that is assembled as part of the compiled program. Literals must be made for large integers, all floats, and most strings.

There are three programs that make literals; each is called with a literal value and a label number:

- `makeilit(i,label)` : integer (not needed for x86)
- `makeflit(i,label)` : float
- `makeblit(i,label)` : byte (string)

A literal is accessed relative to the Instruction Pointer:

```
movsd   .LC4(%rip),%xmml
```

Literals are saved in tables and output at the end of the program.

```
        .align  8
.LC4:
        .long   0
        .long   1078001664
```

Integer Arithmetic Instructions

These instructions operate on registers or memory. `S,D` represent source and destination.

`addl`	`S,D`	$D + S \rightarrow D$
`subl`	`S,D`	$D - S \rightarrow D$
`imull`	`S,D`	$D * S \rightarrow D$
`ldiv`	`S,D`	$D/S \rightarrow D$
`cmpl`	`S,D`	*compare* $D - S$, *set condition*
`andl`	`S,D`	$D \wedge S \rightarrow D$
`orl`	`S,D`	$D \vee S \rightarrow D$
`notl`	`D`	$\neg D \rightarrow D$
`negl`	`D`	$-D \rightarrow D$

Note that arithmetic can be done directly on memory: `i := i + 1` can be one instruction:

```
addl $1,-32(%rbp)
```

Compare and Jump

A *compare* is a subtract that does not store its results; however, the results set the *condition code*, which can be tested by jump instructions.

`cmpl`	`S,D`	*compare* $D - S$, *set condition*, integer
`cmpq`	`S,D`	*compare* $D - S$, *set condition*, pointer
`cmpsd`	`S,D`	*compare* $D - S$, *set condition*, float

The jump instructions test the condition code:

`jmp`	Jump always.
`jle`	Jump if $D \leq S$
`je`	Jump if $D = S$
`jne`	Jump if $D \neq S$
`jge`	Jump if $D \geq S$
`jl`	Jump if $D < S$
`jg`	Jump if $D > S$

Floating Point

These instructions operate on registers or memory. `S,D` represent source and destination.

`addsd`	`S,D`	$D + S \rightarrow D$
`subsd`	`S,D`	$D - S \rightarrow D$
`mulsd`	`S,D`	$D * S \rightarrow D$
`divsd`	`S,D`	$D/S \rightarrow D$
`cmpsd`	`S,D`	*compare* $D - S$, *set condition*

Routines are provided to generate the instruction sequences for `fix`, `float` and `negate` operations.

Intrinsic Functions

Some things that are specified as functions in source code should be compiled in-line. These include:

1. Type-change functions that act as the identity function: `boole`, `ord`, `chr`.
2. Functions that are only a few instructions: `pred` (- 1), `succ` (+ 1), `abs`.
3. Functions that are implemented in hardware: `sqrt` may be an instruction.

Function Calls

For external functions, it is necessary to:

1. Set up the arguments for the function call.
2. Call the function.
3. Retrieve the result and do any necessary final actions.

A function call involves the following:

1. Load arguments into registers:
   - For string literals, address in `%edi`:

     ```
     movl  $.LC12,%edi # addr of literal .LC12
     ```

   - For floating arguments, `%xmm0`
2. Execute a `call` instruction:

   ```
   call   sin
   ```

3. Floating results are returned in `%xmm0`. Integer results are returned in `%eax` or `%rax`.

Volatile Registers

By convention, some registers may be designated:

- **volatile** or **caller-saved**: assumed to be destroyed by a subroutine call.
- **non-volatile** or **callee-saved**: preserved (or not used) by a subroutine.

We will try to use only the registers `%eax`, `%ecx`, and `%edx`, since `%ebx` is callee-saved.

Any floating values that need to be preserved across a call must be saved on the stack prior to the call and restored afterwards. Routines are provided to save one floating register on the stack and restore it.

Details of Function Call

1. For each argument, use `genarith` to compute the argument. If needed, move the result from the register returned by `genarith` to `%xmm0` and mark the `genarith` register unused.
2. For each volatile register that is in use, save it
3. Call the function
4. For each volatile register that is in use, restore it
5. Return the function result register (`%xmm0`, `%eax` or `%rax`) as the result of `genarith`.

IF Statement Generation

Code for an intermediate code statement of the form `(if c p1 p2)` can be generated as follows:

1. Generate code for the condition `c` using the arithmetic expression code generator. Note that a `cmp` instruction should be generated for all comparison operators, regardless of which comparison is used.
2. Generate the appropriate jump-on-condition instruction, denoted `jmp c` below, by table lookup depending on the comparison operator.

```
    jmp c    .L1
    p2               # "else"
    jmp      .L2
.L1:
    p1               # "then"
.L2:
```

The following jump table can be used:

op	$=$	$\neq$	$<$	$\leq$	$\geq$	$>$
c	je	jne	jl	jle	jge	jg
-c	jne	je	jge	jg	jl	jle

IF Statement Optimization

Special cases of `IF` statements are common; these can be compiled as shown below, where `jmp c` represents a jump on condition and `jmp -c` represents a jump on the opposite of a condition.

```
(if c (goto l))                 jmp c   l

(if c (progn) (goto l))         jmp -c  l

(if c p1 (goto l))              jmp -c  l
                                p1

(if c (goto l) p2)              jmp c   l
                                p2

(if c p1)                       jmp -c  L1
                                p1
                            L1:

(if c (progn) p2)               jmp c   L1
                                p2
                            L1:
```

Array References

Suppose the following declarations have been made:

```
var i: integer; x: array[1..100] of real;
```

Assume that `i` has an offset of `4` and `x` has an offset of `8` (since `x` is `double`, its offset must be 8-aligned.). The total storage is `808`. A reference `x[i]` would generate the code:

```
(AREF X (+ -8 (* 8 I)))
```

The effective address is: `%rbp`, minus stack frame size, plus the offset of `x`, plus the expression `(+ -8 (* 8 I))`.

Easy Array References

```
(AREF X (+ -8 (* 8 I)))
```

One way to generate code for the array reference is to:

- use `genarith` to generate `(+ -8 (* 8 I))` in register (`%eax`) (move the result to `%eax` if necessary).
- Issue the instruction `CLTQ` (Convert Long To Quad), which sign-extends `%eax` to `%rax`.
- access memory from the offset and sum of the registers.

```
movsd   %xmm0,-1296(%rbp,%rax)  #  ac[]
```

This is easy from the viewpoint of the compiler writer, but it generates many instructions, including a possibly expensive multiply.

Better Array References

```
(AREF X (+ -8 (* 8 I)))
```

A better way generate the array reference is to:

1. combine as many constants as possible
2. replace the multiply with a shift

Note that in the expression `(+ -8 (* 8 I))` there is an additive constant of `-8` and that the multiply by `8` can be done in the x86 processor by a shift of 3 bits, which can be done for free by the instruction.

This form of code can be generated as one instruction on x86, assuming that `i` is in `%rax`:

```
movsd  %xmm0,-208(%rbp,%rax,8)
```

Pointer References

A pointer operator specifies indirect addressing. For example, in the test program, the code `johnˆ.favorite` produces the intermediate code:

```
(aref (ˆ john) 32)
```

Note that a pointer operator can occur in Pascal *only* as the first operand of an **aref**, and in this case the offset is usually a constant. Compiling code for it is simple: the address is the sum of the pointer value and the offset:

```
movq -1016(%rbp),%rcx  #  john -> %rcx
movl %eax,32(%rcx)     #  ˆ. []
```

This example shows a store of `%eax` into memory at a location 32 bytes past the pointer value in the variable `john`.

`switch` Statement

The `switch` statement is usually evil:

- generates lots of code (lots of `if` statements)
- takes time to execute
- poor software engineering.

```
int vowel(ch)
  int ch;
  {  int sw;
     switch ( ch )
       {  case 'A':  case 'E':  case 'I':
          case 'O':  case 'U':  case 'Y':
            sw = 1; break;
          default: sw = 0; break;
          }
     return (sw);
   }
```

`switch` Statement Compiled

```
vowel:
        save    %sp,-104,%sp
        st      %i0,[%fp+68]
.L14:
        ba      .L16
        nop
.L17:
.L18:
.L19:
.L20:
.L21:
.L22:
        mov     1,%o0
        ba      .L15
        st      %o0,[%fp-8]
.L23:           !    default: sw = 0; break;
        ba      .L15
        st      %g0,[%fp-8]
.L16:
        ld      [%fp+68],%o0
        cmp     %o0,79
        bge     .L_y0
        nop
        cmp     %o0,69
        bge     .L_y1
        nop
        cmp     %o0,65
        be      .L17
        nop
        ba      .L23
        nop
.L_y1:
        be      .L18
        nop
        ... 20 more instructions
.L24:
.L15:
        ld      [%fp-8],%i0
        jmp     %i7+8
        restore
```

`switch` Statement Compiled `-O`

```
[ ... big table constructed by the compiler ... ]
vowel:
        sub     %o0,65,%g1
        cmp     %g1,24
        bgu     .L77000008
        sethi   %hi(.L_const_seg_900000102),%g2
.L900000107:
        sll     %g1,2,%g1
        add     %g2,%lo(.L_const_seg_900000102),%g2
        ld      [%g1+%g2],%g1
        jmpl    %g1+%g2,%g0
        nop
.L77000007:
        or      %g0,1,%g1
        retl    ! Result =  %o0
        or      %g0,%g1,%o0
.L77000008:
        or      %g0,0,%g1
        retl    ! Result =  %o0
        or      %g0,%g1,%o0
```

Table Lookup

```
static int vowels[]
   = {1,0,0,0,1,0,0,0,1,0,0,0,0,
      0,1,0,0,0,0,0,1,0,0,0,1,0};

int vowel(ch)
  int ch;
  {  int sw;
     sw = vowels[ch - 'A'];
     return (sw);
   }
```

Table Lookup Compiled

```
vowel:
        save    %sp,-104,%sp
        st      %i0,[%fp+68]
.L15:
        ld      [%fp+68],%o0
        sll     %o0,2,%o1
        sethi   %hi(vowels-260),%o0
        or      %o0,%lo(vowels-260),%o0
        ld      [%o1+%o0],%i0
        st      %i0,[%fp-8]

        jmp     %i7+8
        restore
```

Table Lookup Compiled -O

```
vowel:
        sll     %o0,2,%g1
        sethi   %hi(vowels-260),%g2
        add     %g2,%lo(vowels-260),%g2
        retl                        ! Result =  %o0
        ld      [%g1+%g2],%o0       ! volatile
```

Bottom Line:

`switch`	46
`switch -O`	15
Table Lookup	10
Table Lookup `-O`	5

Table Lookup beats the `switch` statement in code size and performance; it is also better Software Engineering.

Parameter Passing

Several methods of passing parameters between calling program and subroutine are used:

1. **Call by Reference:** The *address* of the parameter is passed. The storage in the calling program is used (and possibly modified). Used by Fortran, Pascal for `var` parameters, C for arrays.
2. **Call by Value:** The *value* of the parameter is copied into the subroutine. Modifications are not seen by the caller. Expensive for large data, e.g. arrays. Used in Pascal, Java, C for basic types.
3. **Call by Value - Result:** The value of the parameter is copied into the subroutine, and the result is copied back upon exit.
4. **Call by Name:** The effect is that of a *textual substitution* or macro-expansion of the subroutine into the caller's environment. Trouble-prone, hard to implement, slow. Used in Algol.
5. **Call by Pointer:** A pointer to the parameter value is passed. Used in Lisp, in Java for reference types. The object pointed to can be changed.

Macros

A *macro* is a function from code to code, usually turning a short piece of code into a longer code sequence.

Lisp macros produce Lisp code as output; this code is executed or compiled.

```
(defun neq (x y) (not (eq x y)))

(defmacro neq (x y) (list 'not (list 'eq x y)))

(defmacro neq (x y) `(not (eq ,x ,y)))
```

In C, `#define` *name pattern* specifies a textual substitution. If *pattern* contains an operation, it should be parenthesized.

```
#define sum  x + y          /* needs parens */

   z = sum * sum;
```

In-line Compilation

In-line or *open* compilation refers to compile-time expansion of a subprogram, with substitution of arguments, in-line at the point of each call.

Advantages:

- Eliminates overhead of procedure call
- Can eliminate method lookup in an object-oriented system
- Can expose opportunities for optimization across the procedure call, especially with OOP: more specific types become exposed.
- Relative saving is high for small procedures

Disadvantages:

- May increase code size

Optimization

Program optimization can be defined as follows:

> Given a program P, produce a program P' that produces the same output values as P for a given input, but has a lower cost.

Typical costs are execution time and program space. Most optimizations target time; fortunately, the two usually go together.

Optimization is an *economic* activity:

- **Cost:** a larger and sometimes slower compiler.
- **Benefit:**

```
Amount saved by the code improvement
  * number of occurrences in code
  * number of repetitions in execution
  * number of uses of the compiled code
```

It is not possible to optimize everything. The goal is to find *leverage*: cases where there is a large expected payoff for a small cost.

Correctness of Optimization

Optimization must never introduce compiler-generated errors! A program that runs faster but produces incorrect results is not an improvement.

There are often cases where an optimization will *nearly always* be correct.

```
if ( x * n == y * n ) ...
```

might be optimized to:

```
if ( x == y ) ...
```

Is this correct?

In general, one must be able to *prove* that an optimized program will *always* produce the same result.

Optional Optimization

Some compilers either allow the optimizer to be turned off, or require that optimization be requested explicitly.

Reasons for turning optimization off:

- Compilation may be faster.
- If the optimizer produces errors, the errors can be avoided.

With some sophisticated compilers, the users normally turn the optimizer off! This is because the optimizer has the reputation of generating incorrect code.

Optimizations that don't take much compilation time and are guaranteed to be correct should probably be done every time. A slightly longer compilation time is almost always compensated by faster execution.

Local and Global Optimization

Local optimization is that which can be done correctly based on analysis of a small part of the program.
Examples:

- Constant folding: $2 * 3.14 \rightarrow 6.28$
- Reduction in strength: $x * 2 \rightarrow x + x$
- Removing branches to branches:

```
L1:   Goto  L2
```

Global optimization requires information about the whole program to be done correctly.
Example:

```
I * 8          ==>      R1 = I * 8

...                     ...

I * 8          ==>      R1
```

This is correct only if `I` is not redefined between the two points. Doing optimization correctly requires program analysis: a special-purpose proof that program P' produces the same output values as P.

Easy Optimization Techniques

Some good optimization techniques include:

1. Generation of good code for common special cases, such as `i = 0`. These occur frequently enough to provide a good savings, and testing for them is easy.
2. Generation of good code for subscript expressions.
   - Code can be substantially shortened.
   - Subscript expressions occur frequently.
   - Subscript expressions occur inside loops.
3. Assigning variables to registers.
   - Much of code is loads and stores. A variable that is in a register does not have to be loaded or stored.
   - Easy case: assign a loop index variable to a register inside the loop.
   - General case: graph coloring for register assignment.
4. Reduction in strength: `x * 8` $\rightarrow$ `x << 3`

Constant Folding

Constant folding is performing operations on constants at compile time:

```
x = angle * 3.1415926 / 180.0

y = sqrt(2.0)

z = a[3]
```

The savings from doing this on programmer expressions is minor. However, there can be major savings by optimizing:

1. Subscript expressions generated by the compiler.
2. Messages in an object-oriented language.

Constant folding should not allow the programmer to "break" the compiler by writing a bad expression: `sqrt(-1.0)`.

Peephole Optimization

Certain kinds of redundancies in generated code can be corrected by making a linear pass over the output code and transforming patterns seen within a small local area (called the *peephole*):

```
        STA     X
        LDA     X


        JMP     L17
L17:
```

However,

> *It is more blessed to generate good code in the first place than to generate bad code and then try to fix it.*

Loop Unrolling

Loop unrolling is the compile-time expansion of a loop into repetitions of the code, with the loop index replaced by its value in each instance.

```
for (i = 0; i < 3; i++)  disp[i] = c2[i] - c1[i];
```

is expanded into:

```
disp[0] = c2[0] - c1[0];
disp[1] = c2[1] - c1[1];
disp[2] = c2[2] - c1[2];
```

	Loop:	Unrolled:
Instructions:	20	12
Executed:	57	12

The second form runs faster, and it may generate less code. This is a useful optimization when the size of the loop is known to be a small constant at compile time.

Modulo unrolling unrolls a loop modulo some chosen constant. This can significantly reduce the loop overhead without expanding code size too much.

Partial Evaluation

Partial evaluation is the technique of evaluating those parts of a program that can be evaluated at compile time, rather than waiting for execution time.

For example, the rotation of a point in homogeneous coordinates by an angle θ around the x axis is accomplished by multiplying by the matrix:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Many of the cycles consumed in the matrix multiply would be wasted because they would be trivial computations (*e.g.*, multiplying by `1` or adding `0`).

By unrolling the loops of matrix multiply, substituting the values from the coefficient matrix, and performing partial evaluation on the result, a specialized version of the matrix multiply can be obtained. This version saves many operations:

Version:	Load	Store	Add/Sub	Mul	Total
General	128	16	48	64	256
Specialized	24	16	8	16	64

Partial Evaluation[37]

Partial evaluation specializes a function with respect to arguments that have known values. Given a program $P(x, y)$ where the values of variables x are constant, a specializing function `mix` transforms $P(x, y) \rightarrow P_x(y)$ such that $P(x, y) = P_x(y)$ for all inputs y. $P_x(y)$ may be shorter and faster than $P(x, y)$. We call x *static data* and y *dynamic data*.

Partial evaluation involves:

- *precomputing* constant expressions involving x,
- *propagating* constant values,
- *unfolding* or *specializing* recursive calls,
- *reducing* symbolic expressions such as $x * 1$, $x * 0$, $x + 0$, $(if\ true\ S_1\ S_2)$.

A good rule of thumb is that an interpreted program takes ten times as long to execute as the equivalent compiled program. Partial evaluation removes interpretation by increasing the *binding* between a program and its execution environment.

[37]Neil D. Jones, Carsten K. Gomard, and Peter Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993; *ACM Computing Surveys*, vol. 28, no. 3 (Sept. 1996), pp. 480-503.

Example

Suppose we have the following definition of a function `power(x,n)` that computes x^n :

```
(defun power (x n)
  (if (= n 0)
      1
      (if (evenp n)
          (square (power x (/ n 2)))
          (* x (power x (- n 1))))))
```

If this is used with a constant argument `n`, as is often the case, the function can be partially evaluated into more efficient code:

```
(gldefun t3 ((x real)) (power x 5))

(glcp 't3)
result type: REAL
(LAMBDA (X) (* X (SQUARE (SQUARE X))))
```

The recursive function calls and interpretation (`if` statements) have been completely removed; only computation remains. Note that the constant argument `5` is gone and has been converted into control.

Simple Partial Evaluator

```
(defun mix (code env)
  (let (args test fn)
    (if (constantp code)              ; a constant
        code                          ;   evaluates to itself
        (if (symbolp code)            ; a variable
            (if (assoc code env)      ;   bound to a constant
                (cdr (assoc code env)) ; evals to that consta
                code)                  ; else to itself
            (if (consp code)
                (progn
                  (setq fn (car code))
                  (if (eq fn 'if)          ; if is handled
                      (progn               ;     specially
                        (setq test (mix (cadr code) env))
                        (if (eq test t)             ; if true
                            (mix (caddr code) env) ; then par
                            (if (eq test nil)      ; if false
                                (mix (cadddr code) env) ; els
                                (cons 'if
                                  (cons test
                                    (mapcar #'(lambda (x)
                                                (mix x env))
                                            (cddr code))))))))
```

Simple Partial Evaluator...

```
      (progn                                  ; (fn args)
        (setq args (mapcar #'(lambda (x)
                               (mix x env)) ; mix the args
                           (cdr code)))
        (if (and (every #'constantp args)   ; if all constant
                 (not (member fn '(print    ; and no
                        prin1 princ error   ; compile-time
                        format))))          ; side-effects
            (kwote (eval (cons fn args)))   ; eval it now
            (if (and (some #'constantp args); if some constan
                     (fndef fn))            ;  & symbolic fn
                (fnmix fn args)             ;  unfold the fn
                (fnopt (cons fn args)))))))) ; optimize result

(cons 'bad-code code)) ) ) ))
```

Examples

```
>(load "/u/novak/cs394p/mix.lsp")
>(mix 'x '((x . 4)))
4
>(mix '(if (> x 2) 'more 'less) '((x . 4)))
'MORE

(defun power (x n)
  (if (= n 0)
      1
      (if (evenp n)
          (square (power x (/ n 2)))
          (* x (power x (- n 1)))) ) )

>(fnmix 'power '(x 3))
(* X (SQUARE X))

>(specialize 'power '(x 3) 'cube)
>(fndef 'cube)
(LAMBDA (X) (* X (SQUARE X)))
> (cube 4)
64

>(fnmix 'power '(x 22))
(SQUARE (* X (SQUARE (* X (SQUARE (SQUARE X))))))
```

Examples

```
; append two lists
(defun append1 (l m)
  (if (null l)
      m
      (cons (car l) (append1 (cdr l) m))))


>(fnmix 'append1 '('(1 2 3) m))
(CONS 1 (CONS 2 (CONS 3 M)))
```

Binding-Time Analysis

Binding-time analysis determines whether each variable is static (S) or dynamic (D).

- Static inputs are S and dynamic inputs are D.
- Local variables are initialized to S.
- Dynamic is contagious: if there is a statement
  $v = f(...D...)$
  then v becomes D.
- Repeat until no more changes occur.

Binding-time analysis can be *online* (done while specialization proceeds) or *offline* (done as a separate preprocessing phase). Offline processing can *annotate* the code by changing function names to reflect whether they are static or dynamic, e.g. `if` becomes `ifs` or `ifd`.

Futamura Projections[38]

Partial evaluation is a powerful unifying technique that describes many operations in computer science.

We use the notation $[\![\mathtt{P}]\!]_{\mathtt{L}}$ to denote running a program `P` in language `L`. Suppose that `int` is an interpreter for a language `S` and `source` is a program written in `S`. Then:

- $$\begin{aligned} \mathtt{output} &= [\![\mathtt{source}]\!]_{\mathtt{s}}[\mathtt{input}] \\ &= [\![\mathtt{int}]\!][\mathtt{source}, \mathtt{input}] \\ &= [\![[\![\mathtt{mix}]\!][\mathtt{int}, \mathtt{source}]]\!][\mathtt{input}] \\ &= [\![\mathtt{target}]\!][\mathtt{input}] \end{aligned}$$

  Therefore, $\mathtt{target} = [\![\mathtt{mix}]\!][\mathtt{int}, \mathtt{source}]$.

- $$\begin{aligned} \mathtt{target} &= [\![\mathtt{mix}]\!][\mathtt{int}, \mathtt{source}] \\ &= [\![[\![\mathtt{mix}]\!][\mathtt{mix}, \mathtt{int}]]\!][\mathtt{source}] \\ &= [\![\mathtt{compiler}]\!][\mathtt{source}] \end{aligned}$$

  Thus, $\mathtt{compiler} = [\![\mathtt{mix}]\!][\mathtt{mix}, \mathtt{int}] = [\![\mathtt{cogen}]\!][\mathtt{int}]$

- Finally, $\mathtt{cogen} = [\![\mathtt{mix}]\!][\mathtt{mix}, \mathtt{mix}] = [\![\mathtt{cogen}]\!][\mathtt{mix}]$ is a *compiler generator*, i.e., a program that transforms interpreters into compilers.

[38] Y. Futamura, "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler", *Systems, Computers, Controls*, 2(5):45-50, 1971. The presentation here follows Jones *et al.*

Interpreter

This program is an interpreter for arithmetic expressions using a simulated stack machine.

```
(defun topinterp (exp)   ; interpret, pop result
  (progn (interp exp)
         (pop *stack*)))

(defun interp (exp)
  (if (consp exp)                ; if op
      (if (eq (car exp) '+)
          (progn (interp (cadr exp))  ; lhs
                 (interp (caddr exp)) ; rhs
                 (plus))              ; add
          (if ...))      ; other ops
      (pushopnd exp)))   ; operand

(defun pushopnd (arg) (push arg *stack*))

(defun plus ()
  (let ((rhs (pop *stack*)))
    (pushopnd (+ (pop *stack*) rhs))))

>(topinterp '(+ (* 3 4) 5))
17
```

Specialization

The interpreter can be specialized for a given input expression, which has the effect of compiling that expression.

```
>(topinterp '(+ (* 3 4) 5))
17

>(specialize 'topinterp
             '('(+ (* a b) c))
             'expr1 '(a b c))

>(pp expr1)

(LAMBDA-BLOCK EXPR1 (A B C)
  (PROGN
    (PUSH A *STACK*)
    (PUSH B *STACK*)
    (TIMES)
    (PUSH C *STACK*)
    (PLUS)
    (POP *STACK*)))

>(expr1 3 4 5)
17
```

Parameterized Programs

A highly parameterized program is easier to write and maintain than many specialized versions for different applications, but may be inefficient.

Example: Draw a line: (x_1, y_1) to (x_2, y_2).
Options include:

- Width of line (usually 1)
- Color
- Style (solid, dashed, etc.)
- Ends (square, beveled)

If all of these options are expressed as parameters, it makes code longer, makes calling sequences longer, and requires interpretation at runtime. Partial evaluation can produce efficient specialized versions automatically.

Pitfalls of Partial Evaluation

There are practical difficulties with partial evaluation:

- To be successfully partially evaluated, a program must be written in the right way. There should be good *binding time separation*: avoid mixing static and dynamic data (which makes the result dynamic).

```
(lambda (x y z)
  (+ (+ x y) z))
```

```
(lambda (x y z)
  (+ x (+ y z)))
```

- The user may have to give advice on when to unfold recursive calls. Otherwise, it is possible to generate large or infinite programs.

  One way to avoid this is to require that recursively unfolding a function call must make a constant argument smaller according to a well-founded ordering. Branches of dynamic `if` statements should not be unfolded.

Pitfalls ...

- Repeating arguments can cause exponential computation duplication: [39]

```
(defun f (n)
  (if (= n 0)
      1
      (g (f (- n 1)) ) ) )

(defun g (m) (+ m m))
```

- The user should not have to understand the logic of the output program, nor understand how the partial evaluator works.
- Speedup of partial evaluation should be predictable.
- Partial evaluation should deal with typed languages and with symbolic facts, not just constants.

[39] Jones *et al.*, p. 119.

Program Analysis

To correctly perform optimizations such as moving invariant code out of loops or reusing common subexpressions, it is necessary to have *global* information about the program.[40]

Control flow analysis provides information about the potential control flow:

- Can control pass from one point in the program to another?
- From where can control pass to a given point?
- Where are the loops in the program?

Data flow analysis provides information about the definition and use of variables and expressions. It can also detect certain types of programmer errors.

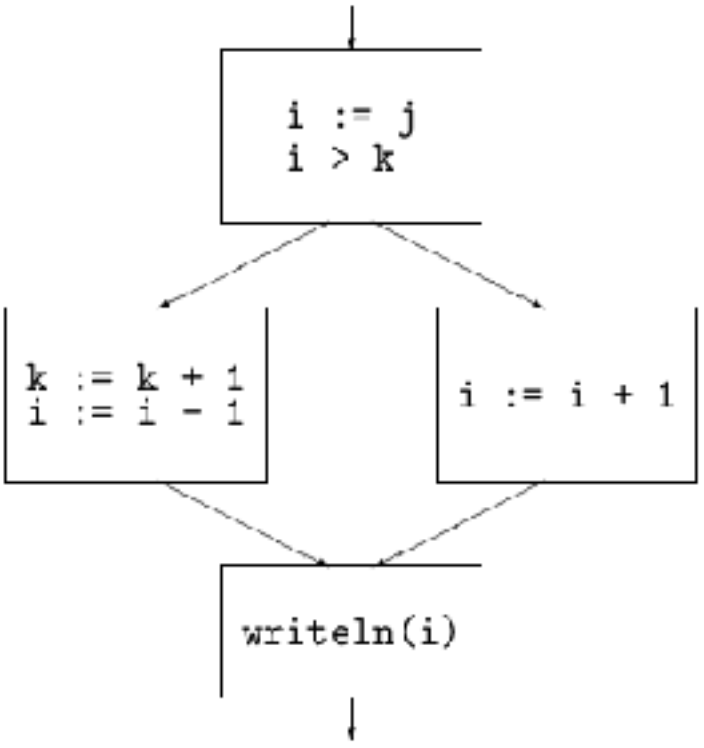
- Where is the value of a variable assigned?
- Where is a given assignment used?
- Does an expression have the same value at a later point that it had at an earlier point?

[40]This treatment follows Marvin Schaefer, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, 1973.

Basic Block

A *basic block* (or *block* for short) is a sequence of instructions such that if any of them is executed, all of them are. That is, there are no branches in except at the beginning and no branches out except at the end.

```
begin
   i := j;
   if i > k
     then begin k := k + 1; i := i - 1 end
     else i := i + 1;
   writeln(i)
end
```



Finding Basic Blocks

Basic blocks are easily found by a compiler while processing a program.

A *leader* is the first statement of a basic block:

1. the first statement of a program
2. any statement that has a label or is the target of a branch
3. any statement following a branch

A basic block is a leader and successive statements up to the next leader.

Note that branch statements themselves do not appear in basic blocks, although the computation of the condition part of a conditional branch will be included.

In a graph representation of a program, basic blocks are the nodes of the graph, and branches are the arcs between nodes.

Relations and Graphs

The *cartesian product* of two sets A and B , denoted $A \times B$, is the set of all ordered pairs (a, b) where $a \in A$ and $b \in B$.

A *relation* between two sets is a subset of their cartesian product.

A *graph* is a pair (S, Γ) where S is a set of nodes and $\Gamma \subseteq S \times S$.

Properties of relations:

Property:	Definition:
Reflexive	$\forall a \quad (a, a) \in R$
Symmetric	$\forall a, b \quad (a, b) \in R \rightarrow (b, a) \in R$
Transitive	$\forall a, b, c \; (a, b) \in R \wedge (b, c) \in R \rightarrow (a, c) \in R$
Antisymmetric	$\forall a, b \quad (a, b) \in R \wedge (b, a) \in R \rightarrow a = b$

A relation that is reflexive, symmetric, and transitive is an *equivalence relation*, which corresponds to a *partition* of the set (a set of disjoint subsets whose union is the set).

A relation that is reflexive, antisymmetric, and transitive is a *partial order*. Example: $\leq$.

Graph Notations

Let (S, Γ) be a graph and $b \in S$ be a node.

$\Gamma b \ = \ \{x \in S \ \mid \ (b, x) \in \Gamma\}$
are the nodes that are *immediate successors* of b .

$\Gamma^{+} b \ = \ \{x \in S \ \mid \ (b, x) \in \Gamma^{+}\}$
are the nodes that are *successors* of b .

$\Gamma^{-1} b \ = \ \{x \in S \ \mid \ (x, b) \in \Gamma\}$
are the nodes that are *immediate predecessors* of b .

Let $A \subset S$ be a subset of the set of nodes S.

$\Gamma A \ = \ \{y \in S \ \mid \ (x, y) \in \Gamma \wedge x \in A\}$
is the set of nodes that are *immediate successors* of nodes in A .

$\Gamma^{-1} A \ = \ \{x \in S \ \mid \ (x, y) \in \Gamma \wedge y \in A\}$
is the set of nodes that are *immediate predecessors* of nodes in A .

We say (A, Γ_A) is a *subgraph* of (S, Γ) , where
$\Gamma_A x \ = \ \Gamma x \cap A$
is the set of transitions within the subgraph.

Bit Vector Representations

Subsets of a finite set can be efficiently represented as bit vectors, in which a given bit position is a `1` if the corresponding item is an element of the subset. Representing a 128-element set takes only 4 32-bit words of memory.

Operations on sets can be done on whole words.

Set operation:	Bit vector operation:
$\in$	$\wedge$ with vector for element or test bit
$\cap$	$\wedge$
$\cup$	$\vee$
set complement of A	$\neg A$
set difference, $A - B$	$A \wedge \neg B$

Operations on the bit vector representation are $O(n/32)$, compared to $O(n \cdot m)$ with other methods.

Example: assign a bit for each program variable or subexpression.

Boolean Matrix Representation of Graph

A relation R or graph on a finite set can be expressed as a boolean matrix M where:

$$M[i,j] = 1 \quad \text{iff} \quad (i,j) \in R \ .$$

Multiplication of boolean matrices is done in the same way as ordinary matrix multiplication, but using $\wedge$ for $\cdot$ and $\vee$ for $+$.

Property:	Matrix:
Identity, R^0	I_n (identity matrix)
Inverse, R^{-1} or Γ^{-1}	M^T
Reflexive	$I \subseteq M$
Symmetric	$M = M^T$
Transitive	$M^2 \subseteq M$
Antisymmetric	$M \cap M^T \subseteq I_n$
Paths of length n	M^n
Transitive closure Γ^+	$\bigcup_{i=1}^{n} M^i$
Reflexive transitive closure Γ^*	$\bigcup_{i=0}^{n} M^i$

Example: Let the set S be basic blocks of a program and Γ be transfers of control between blocks.

Dominators

Let e denote the first block of a program. A node d *dominates* a node n iff every simple path from e to n passes through d .

For a given node n, its *immediate dominator* is the dominator closest to it. A tree structure is formed by immediate dominators, with e being the root of the tree.

A loop header h dominates all the nodes in the loop. A *back edge* is an edge $n \rightarrow h$ where h dominates n.

Intervals

An *interval* is a subgraph that basically corresponds to a program loop.

An interval I with initial node h is the maximal subgraph (I, Γ_I) of (S, Γ) such that:

1. $h \in I$
2. $x \in I \rightarrow x \in \Gamma^* h$
3. $I - \{h\}$ is cycle-free
4. if $x \in I - \{h\}$, then $\Gamma^{-1} x \subset I$.

To construct an interval starting with node h:

1. initially, set $I := \{h\}$
2. repeat $I := I \cup \{x \in \Gamma I \mid \Gamma^{-1} x \subseteq I\}$
   until there are no more additions.

Members of $\Gamma I - I$ must be the heads of other intervals.

Definition and Reference of Variables

We assume that each variable is assigned a unique *bit number* so that it can be used in bit vectors. Likewise, each compiler variable or subexpression $\alpha \leftarrow a \circ b$ is assigned a bit number.

A variable is *defined* each time it is assigned a value. A variable is *referenced* (*used*) whenever its value is read.

The statement `x := a * b` first references `a` and `b` and then defines `x`.

The statement `x := x + 1` references `x` and then defines `x`.

A computation $a \circ b$ is *redundant* if its value is available in some variable α.

A subexpression is *computed* whenever it appears in an expression. A subexpression is *killed* if any of its components is defined or killed.

The statement `x[i*3] := a * b` computes `a * b` and `i * 3` and kills `x[`*anything*`]` .

Data Flow Analysis for a Block

Computed and *killed* vectors for a basic block can be found as follows:

- initially, $comp := \emptyset$ and $kill := \emptyset$.
- for each statement $v := a \circ b$ where $\alpha \leftarrow a \circ b$
  1. $comp := comp \cup \{\alpha\}$
  2. $kill := kill \cup kill_v$
  3. $comp := (comp - kill_v) \cup \{v\}$

  where $kill_v$ is the set of all expressions involving v directly or indirectly and $(comp - kill_v)$ is set difference.

Example: `I := I + 1`
This statement first computes the expression `I + 1`, but then it kills it because it redefines `I`.

Availability of Expressions

The expression $\alpha \leftarrow a \circ b$ is *available at a point* p if the value of the variable α is the same as the value of $a \circ b$ computed at the point p.

The expression α is *available on entry* to block b iff α is available on exit from all immediate predecessors of b.

$$avail_{entry}(b) \;=\; \cap_{x \in \Gamma^{-1} b}\, avail_{exit}(x)$$

The expression α is *available on exit* from block b iff α is available at the last point of b.

$$avail_{exit}(b) \;=\; (avail_{entry}(b) - kill(b)) \cup comp(b)$$

In general, a system of simultaneous boolean equations may have multiple consistent solutions. It is necessary to compute the *maximal solution* of the set of boolean equations for intervals at all levels of the derived graph.

Data Flow Analysis for an Interval

If the expressions that are available on entry to the head of the interval are known, the values for all blocks in the interval can be computed.

For each block b whose predecessors have had their values computed,

$$avail_{entry}(b) \;=\; \Pi_{x \in \Gamma^{-1} b}\; avail_{exit}(x)$$
$$avail_{exit}(b) \;=\; avail_{entry}(b) \cdot not(kill(b)) + comp(b)$$

No expressions are available on entry to the first block of a program.

Busy Variables

A dual notion to *available* is *busy*.

A variable is *busy* or *live* if it will be used before being defined again; otherwise, it is *dead*.

A variable is *busy on entrance* to a block b if it is used in block b before being defined, or if it is not defined or killed in block b and is busy on exit from b .

A variable is *busy on exit* from a block b if it is busy on entry to any successor of b .

We can define a bit vector *referenced*, meaning that an expression is referenced in a block before being computed or killed, and solve equations for *busy on entrance* and *busy on exit* in a manner analogous to that for the *available* equations.

Variable Uses and Register Assignment

A *def-use chain* is the connection between a definition of a variable and the subsequent use of that variable. When an expression is computed and is *busy*, the compiler can save its value. When an expression is needed and is *available*, the compiler can substitute the compiler variable representing its previously computed value.

Register allocation can be performed by graph coloring. A graph is formed in which nodes are def-use chains and (undirected) links are placed between nodes that share parts of the program flow graph.

A graph is *colored* by assigning "colors" to nodes such that no two nodes that are linked have the same color. Colors correspond to registers.

Register Allocation by Graph Coloring

An undirected graph is *colored* by assigning a "color" to each node, such that no two nodes that are connected have the same color.

Graph coloring is applied to register assignment in the following way:

- Nodes of this graph correspond to variables or subexpressions.
- Nodes are connected by arcs if the variables are busy at the same time.
- Colors correspond to registers.

A heuristic algorithm is applied to find approximately the minimum number of colors needed to color the graph. If this is more than the number of available registers, *spill code* is added to reduce the number of colors needed.

By keeping as many variables as possible in registers, the code can be significantly improved.

Overview of Global Optimization

A globally optimizing compiler will perform the following operations:

1. Perform interval analysis and compute the derived graphs.
2. Order nodes using an ordering algorithm to find dominators.
3. Find basic available and busy information for blocks.
4. Solve boolean equations to get available and busy information for each block.
5. Replace common subexpressions by corresponding compiler variables.
6. Assign registers using graph coloring.

The information provided by data flow analysis provides a special-purpose proof that the optimized program is correct (produces the same answers).

`gcc` Compiler Optimization Options [41]

- `-O` Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

  Without '-O', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

  Without '-O', only variables declared register are allocated in registers.

  With '-O', the compiler tries to reduce code size and execution time.

- `-fforce-mem` Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load.

- `-fforce-addr` Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as '-fforce-mem' may.

- `-finline` Pay attention the inline keyword. Normally the negation of this option '-fno-inline' is used to keep the compiler from expanding any functions inline.

- `-finline-functions` Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

- `-fcaller-saves` Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced.

[41] From the `man gcc` page.

gcc Optimizations

- `-fstrength-reduce` Perform the optimizations of loop strength reduction and elimination of iteration variables.
- `-fthread-jumps` Perform optimizations where we check to see if a jump branches to a location where another comparison subsumed by the first is found. If so, the first branch is redirected to either the destination of the second branch or a point immediately following it, depending on whether the condition is known to be true or false.
- `-funroll-loops` Perform the optimization of loop unrolling. This is only done for loops whose number of iterations can be determined at compile time or run time.
- `-fcse-follow-jumps` In common subexpression elimination, scan through jump instructions in certain cases. This is not as powerful as completely global CSE, but not as slow either.
- `-frerun-cse-after-loop` Re-run common subexpression elimination after loop optimizations has been performed.
- `-fexpensive-optimizations` Perform a number of minor optimizations that are relatively expensive.
- `-fdelayed-branch` If supported for the target machine, attempt to reorder instructions to exploit instruction slots available after delayed branch instructions.
- `-fschedule-insns` If supported for the target machine, attempt to reorder instructions to eliminate execution stalls due to required data being unavailable. This helps machines that have slow floating point or memory load instructions by allowing other instructions to be issued until the result of the load or floating point instruction is required.

Loop Transformations

Sometimes loops can be transformed to different forms that are faster.

```
for i := 1 to 1000 do
  for j := 1 to 1000 do
    x[i,j] := y[i,j];
```

This might be transformed to a single, linear loop:

```
for i := 1 to 1000000 do  x[i] := y[i];
```

Then it might be generated as a block-move instruction.

Code motion is moving code to a more favorable location, *e.g.*, moving invariant code out of loops:

```
for i := 1 to 1000 do
  x[i] := y[i] * sqrt(a);
```

The code `sqrt(a)` does not change within the loop, so it could be moved above the loop and its value reused.

Strip Mining

Getting effective performance from a multi-processor machine (i.e., getting speedup close to n from n processors) is a difficult problem.

For some matrix computations, analysis of loops and array indexes may allow "strips" of the array to be sent to different processors, so that each processor can work on its strip in parallel.

This technique is effective for a significant minority (perhaps 25%) of important matrix computations.

Induction Variable Transformation

Some compilers transform the *induction variable* to allow simplified subscripting expressions:

```
(:= I 1)
(LABEL 1)
(IF (<= I 1000)
    (PROGN ... (AREF X (+ -8 (* 8 I)))
           (:= I (+ I 1))
           (GOTO L1)))
```

might be transformed to:

```
(:= I' 0)
(LABEL 1)
(IF (<= I' 7992)
    (PROGN ... (AREF X I')
           (:= I' (+ I' 8))
           (GOTO L1)))
```

Note that the loop index has no meaning outside the loop and may not have storage assigned to it. Some machines can automatically increment an index register after it is used (called *postincrement*).

Finite Differencing

Finite differencing[42] is a general technique for optimizing expensive computations $f(i)$ that occur in a loop:

- Maintain local variables that hold previous values of the expensive computation $f(i)$ and perhaps some auxiliary values.
- Incrementally compute a new value $f(i + \delta)$ using:
  - the previous value $f(i)$
  - a *difference* from $f(i)$.

Example: $f(i) = i^2$

i	0	1	2	3	4	5
i^2	0	1	4	9	16	25
first difference:		1	3	5	7	9
second difference:		2	2	2	2	

[42] Paige, R. and Koenig, S., "Finite Differencing of Computable Expressions", *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3 (July 1982), pp. 402-454

Example: Computing Squares

Assume that multiplication is expensive. Consider the problem of computing squares of succesive integers.

```
for i := 0 to 99 do
  x[i] := i*i;
```

versus

```
next := 0;
delta := 1;
for i := 0 to 99 do
  begin
     x[i]  := next;
     next  := next + delta;
     delta := delta + 2
  end;
```

The second version has more code, but does no multiplication.

This form of computation has a long history; it was the basis of Babbage's Difference Engine.

General Case

Given an expression $f(x_1, ..., x_n)$, create a variable E to hold its value.

- **Initialize:** Create code

  $$E = f(x_{1_0}, ..., x_{n_0})$$

  to establish E for the initial values of its arguments.

- **Derivative:** Replace each statement dx_i that modifies some variable x_i of E by the statements:

  $$\partial^- E\langle dx_i \rangle$$

  $$dx_i$$

  $$\partial^+ E\langle dx_i \rangle$$

- **Redundant Code Elimination:** replace each occurrence of $f(x_1, ..., x_n)$ by E.

- **Dead Code Elimination:** remove any code that is now unused.

Finite Differencing for Set Operations

Finite differencing can be especially useful in optimizing set operations. Consider the following expression that is used in solving the "k queens" problem:[43]

$$i \notin range(part\_sol) \wedge i \in \{1..k\}$$

This can be transformed to:

$$i \in setdiff(\{1..k\}, part\_sol)$$

A variable $unoccupied\_rows$ can be introduced for this expression. Its initial value is the set $\{1..k\}$.

An update to $part\_sol$ (by recursive call),

$$part\_sol = append(part\_sol, i)$$

leads to a corresponding change to $unoccupied\_rows$

$$unoccupied\_rows = unoccupied\_rows - \{i\}$$

This incremental update may be much cheaper than doing the original range computation or set difference every time.

[43] D.R. Smith, KIDS: A Semiautomatic Program Development System, *IEEE Trans. Software Engineering,* vol. 16, no. 9, Sept. 1990, pp. 1024-1043.

Memoization

Memoization (or *memorization*) is the technique of saving previously calculated values of an expensive function $f(x)$. If a new request to compute $f(x)$ uses a value x that was used previously, the value of $f(x)$ can be retrieved from a table faster than it could be recomputed.

Compare:

- caching
- common subexpression elimination

Advanced CPU's may implement some memoization in hardware: if the CPU can determine that a computation has already been done and exists in a register, it can reuse the result.

Hardware Assistance

Hardly anyone writes programs in assembly language now. The performance of a computer depends on both the hardware and the compiler.

Hardware assistance can greatly improve performance for some expensive program operations:

- subroutine call
- array references
- matrix operations, graphics

Hardware designers increasingly rely on compilers to handle things that used to be done by hardware (e.g. delaying use of a result until it is ready), allowing CPU's to run faster.

PowerPC Features

The PowerPC has several features that the compiler can use to improve the performance of generated code:

- Store-multiple instruction: makes saving registers faster, speeding up subroutine call.
- Multiple condition-code registers: better code can be generated for compound tests.
- Cache prefetch instruction: allows the CPU to start fetching memory before it is needed, avoiding cache miss delays.
- Superscalar architecture: 3 units (integer, float, branch) can execute simultaneously.
- Out-of-order execution: the processor can look ahead in the instruction stream for an instruction to execute. The compiler may move instructions to maximize this effect.
- Conditional branch instructions can indicate the "expected" outcome (branch or not branch). The processor can speculatively execute instructions in the expected path.

SPARC Features

The SPARC architecture has features that improve the performance of generated code:

- *Register windows* on an integer register stack greatly improve the speed of subroutine call:
  - Parameter transmission is easy
  - No need to save and restore integer registers

Hardware Trends

Trends in modern CPU architecture present new demands and opportunities for the compiler writer:

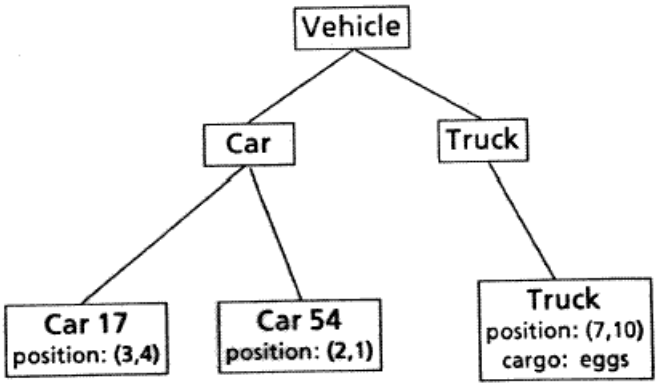
- Larger memories: expansion of code size is not as costly as before; might trade code size for speed.
- The compiler can coexist with the running program, for just-in-time compilation.
- High cost for cache misses: a cache miss can cause the CPU to *stall* (wait) for many execution cycles. Code may be reordered to avoid cache misses.
- Multiple processors: using the processors effectively (getting a speedup of n from n processors) is a difficult challenge. *Strip mining* may be used to send parts of loops (and memory) to different processors.
- Superscalar architecture: code may be re-ordered to take advantage of out-of-order and speculative execution.
- As number of transistors on chip increases, things that were software become hardware: math subroutines, message passing, memoization, dynamic optimization.

Object-oriented Programming

Object-oriented programming (OOP) originated in the SIMULA language (1967) for discrete event simulation. The desire was to simulate large numbers of similar objects in an efficient manner. A class/instance representation achieves this goal.

- **Class:** represents the *behaviors* that are shared by all of its instances. Behaviors are implemented by *methods*.
- **Instance:** represents the data for a particular individual.

Classes are arranged in a hierarchy, with inheritance of methods from higher classes.



Access to Objects

Access to objects is accomplished by sending *messages* to them.

- Retrieving data values: `(send obj x)` `obj.x()`
- Setting data values: `(send obj x: 3)` `obj.setx(3)`
- Requesting actions: `(send obj print)` `obj.print()`

Domain Analysis

A domain analysis in English suggests an object decomposition:

- nouns correspond to objects or variables
- verbs correspond to methods
- adjectives correspond to attributes

An *ontology*, the set of things that exist, is represented by the class hierarchy.

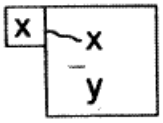
Some have contrasted imperative programming (*subject* oriented) with OOP.

Internal Implementation is Hidden

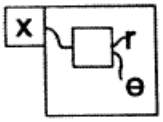
Messages define a standard interface to objects. Objects may have different internal implementations as long as the message interface is maintained. If access is controlled via messages, OOP facilitates *information hiding*: application programs do not know and cannot depend on the internal implementation of the object.

Example: Vector `(send v x)`

- Vector type 1: `x` is stored.



- Vector type 2: `r` and `theta` are stored. `x` is computed as `r * cos(theta)`



The two kinds of vectors appear the same to the outside world.

Encapsulation with OOP

Object-oriented programming provides *encapsulation:* an external *interface* to an object in terms of messages is defined, but the internal implementation of the object is hidden.

Modularity: Objects are often a good way to think about the application domain.

Modifiability: The internal implementation of an object can be changed without modifying any other programs, as long as the external interface is maintained.

Expandability: New kinds of objects can be added to an existing system, as long as they present the same interface as existing objects.

To maintain encapsulation, it is good to make all accesses into messages, avoiding all direct accesses to instance variables: `obj.x()` instead of `obj.x`

- Hides distinction between what is computed and what is stored.
- Improves reusability and maintainability.
- May hurt performance.

Object-oriented Programming Terminology

- *Object:* typically refers to an instance (although classes may be objects too).
- *Class:* a description of a set of similar objects, the instances. This description typically includes:
  - *Instance variable* descriptions: the names and types of variables that are assumed to be defined for every subclass or instance of the class.
  - *Methods:* definitions of messages to which members of the class can respond.
- *Instance:* an individual member of a class. Typically an instance is a member of exactly one class. An instance is a data structure that:
  - can be identified as being an object
  - denotes the class to which the object belongs
  - contains the instance variables
- *Superclass:* a class to which a given class belongs. Sometimes a class may have more than one superclass.

Terminology ...

- *Message:* an indirect procedure call. A message is *sent* to an instance object. The message contains a selector and optional arguments. The selector is looked up in the class of the object (or one of its superclasses) to find the method that implements the message. That method is called with the object to which the message was sent as its first argument.
- *Selector:* the name of a message. A generic procedure name, e.g. `draw`.
- *Method:* the procedure that implements a message. Often the name of a method is the class name hyphenated with the selector, *e.g.* `square-draw`.

Implementation of Objects

An object is basically just a record, with a bit of extra information:

- A *tag* to identify the record as an object. It may be more efficient to tag the class, eliminating the need to tag each object; if the language is strongly typed, runtime tags may not be needed.
- The *class* of the object: a pointer to the class data structure.
- The *instance variables* of the object.

Thus, an object requires only slightly more storage than an ordinary record.

Allocation of space in an object and access to instance variables are the same as for records.

Are Basic Types Objects?

There is an advantage of consistency if basic types such as `integer` and `real` are objects. However:

- A *boxed* number requires more storage. (16 bytes for Java `Integer` vs. 4 bytes for `int`.)
- A statement `i = i + 1` requires allocating a new object (and garbage collecting the old one): a significant performance penalty.

Lisp codes small integers as special pointer values, eliminating the need to box small integers; reals are boxed.

Java caches and reuses small `Integer` values from -128 to 127.

An alternative is to have *both* boxed and unboxed numbers; this improves performance but results in an inconsistent type system. Java uses this method; as a result, a generic method that does comparison may have to be recoded 6 times because of type inconsistencies.

Inheritance and Class Structure

OOP allows subclasses to *extend* a class, both in terms of instance variables and methods.

A common practice it to let subclasses *add* instance variables; the subclass inherits all instance variables of its parent.

- Subclasses simply add variables to the end of the record.
- Methods inherited from a superclass are guaranteed to work, since all needed instance variables will be available at the same constant offsets from the beginning of the record.
- It is not possible to change an object from a superclass to one of its subclasses, since that would increase its size after it is allocated.

Multiple inheritance could cause problems of inconsistent offsets or name conflicts.

Message Sending

Sending a message is just a function call. However, the function (*method*) that is called is determined dynamically by the runtime type of the object to which the message is sent.

Sending a message to an object, `(send obj draw x y)` or `obj.draw(x, y)` involves the following steps:

1. Find the method corresponding to the message selector. If the runtime object is a `circle`, the `draw` method would be looked up as `circle-draw`.
2. Assemble the arguments:
   - The object to which the message was sent (the `self` or `this` argument)
   - Other arguments included in the message
3. Call the method function with the arguments.
4. Return the result returned by the method.

Dynamic Method Lookup

Each class associates a *selector* with the method that implements the selector for that class:

```
((area circle-area)
 (draw circle-draw) ...)
```

A typical dynamic method lookup procedure is:

- Look for the selector in the class of the object to which the message was sent.
- If it is not found, look in superclasses of the class.

The first method found in this search up the class tree is the one that is used. This provides:

- *shadowing:* a more specific subclass method overrides a superclass method with the same name.
- *inheritance:* methods of a superclass are inherited by its subclasses.

Static Method Lookup

In a static language, all of the methods defined for each class can be determined at compile time. Lookup of the called routine can then be done efficiently:

- Selector names are converted to integer offsets.
- A class is implemented during execution as an array of method addresses.
- A method call `obj.sel(args)` is executed by:
  - Getting the class pointer from `obj`
  - Adding the offset associated with `sel`
  - Getting the method address from that address
  - Calling the method.

This takes only a few instructions more than an ordinary function call. This technique is used in C++.

Multiple Inheritance

Multiple inheritance means that a class can have multiple superclasses and inherit from them.

Multiple inheritance can cause several problems:

- Name conflicts of instance variables.
- Methods with the same name from multiple parents.
- Instance variables no longer have simple constant offsets.

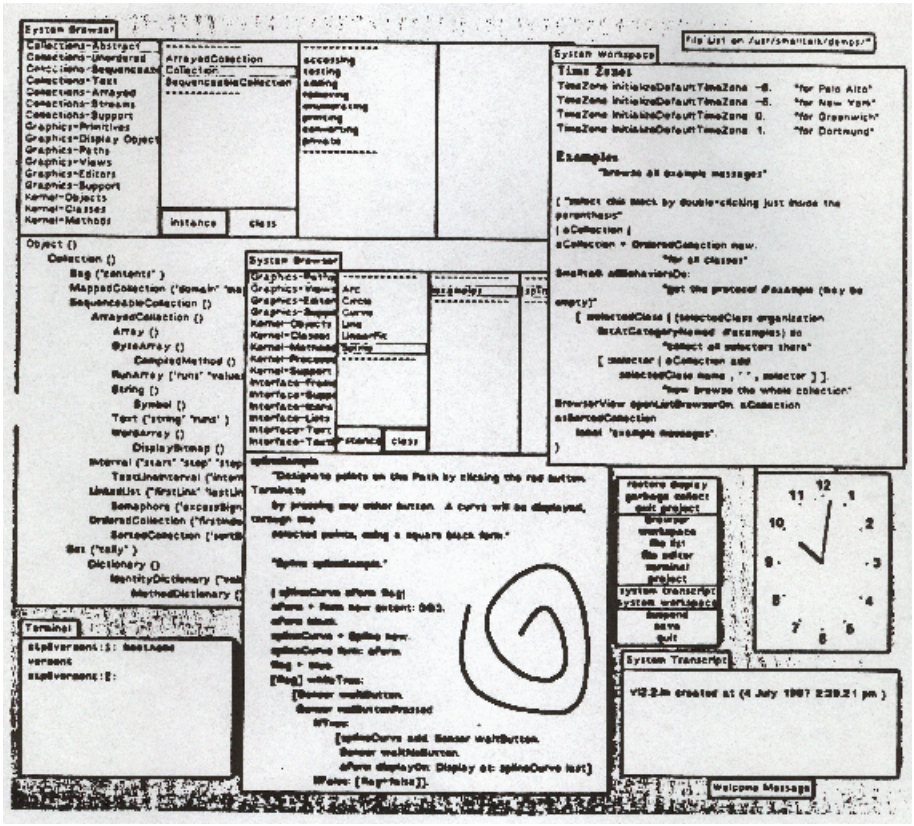
Variations of inheritance include:

- Using types of all arguments (not just the first) to determine the method (CLOS)
- Method Combination (Flavors): the methods of multiple parents can be combined, e.g. a window with border and title.
- SendSuper (Loops): a more specialized subclass method can call the parent method. Useful e.g. to assign serial numbers to manufactured objects.

Improving OOP Efficiency

- Use table lookup (array of method addresses) to make method lookup fast (C++).
- Cache selector and method at lowest class level to avoid search (Self).
- Compile specialized methods "just in time" when first used:
  - `send`s can be converted to function calls since the class is known
  - small methods can be compiled in-line
  - partial evaluation may improve efficiency

Smalltalk



Smalltalk Code

```
|  minLength i j |

  minLength := self length.

  100 timesRepeat:
   [i := ((Float random * self size) truncate + 1)
            asInteger.
    j := ((Float random * self size) truncate + 1)
            asInteger.
    self exchange: i and: j.
    self length < minLength
         ifTrue: [minLength := self length]
         ifFalse: [self exchange: i and: j]]
```

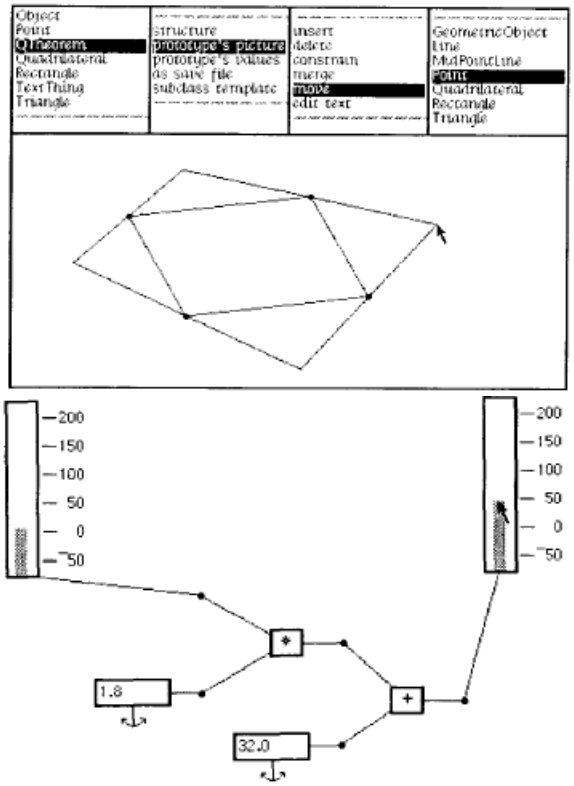
ThingLab

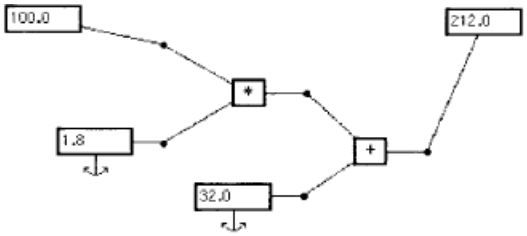
ThingLab[44] is an interactive graphical simulation system based on Smalltalk.

- Objects have ports that can be connected graphically.
- If an object is modified interactively, it propagates changes to its ports, and thus to objects to which it is connected.
- Features of objects can be "anchored" to prevent them from changing.
- An object may do search by attempting to change different port values to make itself consistent.
- Problems are solved by value propagation or by relaxation.

[44]Borning, A., "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory", *ACM Trans. on Programming Languages and Systems*, vol. 3, no. 4 (Oct. 1981), pp. 353-387.

ThingLab Examples





Good Features of OOP

- Encapsulation: implementations of objects can be modified without modifying clients. Data types and related procedures are grouped.
- Polymorphism: some generic methods can be reused effectively.
- Inheritance: methods can be inherited within a hierarchy.

Unfortunate Features of OOP

- OOP tends to require that everything be objects. Hard to use with existing systems.
- Reusing generic procedures is not easy:
  - Programmer must know method name and arguments.
  - Arguments must respond appropriately to all messages that may be sent to them by the method or anything it calls.
  - Methods may produce unexpected and unwanted side-effects.
- Slow in execution.
  - Method lookup overhead
  - Opacity of objects prevents optimization across messages.
  - A layered object structure compounds the problem.

  Some OOP systems "succeed" but have to be discarded and rewritten for efficiency.
- System structure becomes baroque and hard to change (just the opposite of what is advertised).

Why OOP Is Not Enough

- OOP requires the programmer to know too many details about object and method names and implementations.
- OOP requires application data to conform to conventions of the generic procedures:
  - OOP requires an object to *be* a member of a class rather than *be viewable as* a member.
  - Some OOP systems automatically include slots of supers in all their descendants. This inhibits use of different representations.
  - An object cannot be a member of a class in more than one way. (A `pipe` is a `circle` in two ways.)
  - No single definition of an object is likely to encompass all possible uses (or if it did, it would be too big for typical uses).
- Responsibility for a task may be split across many classes.
- OOP is often slow in execution.
- OOP doesn't play well with relational databases.

Top Ten Lies About OOP [45]

10. Objects are good for everything.

9. Object-oriented software is simpler.
(No: everything is a server, and servers are harder to write.)

8. Subclassing is a good way to extend software or libraries.

7. Object-oriented toolkits produce interchangeable components.
(No: you get components the size of a nuclear aircraft carrier, with internal interfaces that are too complex to duplicate.)

6. Using object-oriented programming languages helps build object-oriented systems.

5. Object-oriented software is easier to evolve.
(No: jigsaw-puzzle modularity is a serious problem.)

4. Classes are good for modularity.
(No: most worthwhile modules have more than one class.)

3. Reuse happens.
(No: you have to work too hard to make it possible. He distinguished between *use*, exploiting existing code, and *reuse*, building new code by extending existing code.)

2. Object-oriented software has fewer bugs.
(No: it has different bugs.)

1. C++ is an object-oriented programming language.

[45] From a keynote talk at the June 1993 Usenix conference by Mike Powell of Sun Labs, who is working on an object-oriented operating system in C++ (as recorded by Henry Spencer in the July 1993 issue of ";login:" magazine, whose remarks are in parentheses.)

Aspect-Oriented Programming

Aspect-Oriented Programming[46] is intended to facilitate coding of *cross-cutting* aspects, i.e. those aspects that cut across a typical decomposition of an application in terms of object classes:

- Error handling
- Memory management
- Logging

In Aspect-Oriented Programming, an *aspect weaver* combines program fragments from separate aspects of a design at *cut points* or *join points*.

- **Adv:** Code for the aspect can be described in one place, making it easy to change.
- **Dis:** Combination of source code from different places violates modularity.

AspectJ is an aspect-oriented system for Java.

[46] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, "Aspect-Oriented Programming", in ECOOP '97, vol. 1241 of LNCS, Springer-Verlag, June 1997.

Lisp

Lisp facilitates the manipulation of symbolic expressions such as program code. Therefore, Lisp is ideal for advanced compilation techniques that involve transformation of code.

Greenspun's Tenth Rule of Programming:

> Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.

Example:

Language:	Lines:	Time:	Personnel:	Cost
C++	25,000	10 years	Ph.D.	$2,000,000
Lisp	150	1 week	Freshman	$200

History of Lisp

Lisp was invented in the late 1950's and early 1960's at M.I.T. by John McCarthy. Lisp is based in part on the Lambda Calculus, a mathematical formalism developed by Alonzo Church (1903-1995).

Lisp is the second-oldest language still in widespread use (the oldest is Fortran).

Lisp is especially used for work in Artificial Intelligence. It is also an ideal language for advanced work in compilers.

Scheme is a small, simple dialect of Lisp. Common Lisp is a larger, standardized dialect of Lisp with many features.

Advantages of Lisp

- **Recursion:** A program can call itself as a subroutine.
- **Dynamic Type Checking:** types are tested at runtime
  - Allows short, elegant, powerful programs
  - Incurs a performance penalty
- **Garbage Collection:** Data storage is recycled automatically.
- **Uniform Representation:** Programs and data have the same form.
  - Programs can examine other programs.
  - Programs can write programs.
  - Programs can modify themselves (learn).
  - Data structures can contain programs.
- **Interaction:** The user can combine program writing, compilation, testing, debugging, running in a single interactive session.

Lisp Interaction

The *Lisp Interpreter* interacts with the user:

1. Read an expression from the terminal.
2. Evaluate the expression to determine its value.
3. Print the value; go to step 1.

This is sometimes called the *read-eval-print loop*.

The interpreter prints a *prompt*, which is `>` in GCL [47].

```
% gcl
>57
57

>(+ 32 57)
89

>(+ (* 8 4) (- 60 3))
89

>(sqrt 2)
1.4142135623730951
```

[47] Gnu Common Lisp. Originally developed at the University of Kyoto as Kyoto Common Lisp or KCL, it was enhanced by Prof. Bill Schelter at UT to become Austin Kyoto Common Lisp or AKCL, later renamed GCL. To exit GCL, enter `(bye)`. To get out of an error break, enter `:q` or `:h` for help.

Function Definition

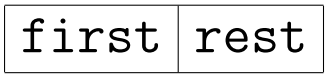
Functions are defined using `defun` (define function):

```
>(defun myabs (x)
   (if (>= x 0)
       x
       (- x) ) )

>(myabs 3)
3

>(myabs -7)
7
```

Local variables can be declared using `let`. Variables can be assigned values using `setq` (set-quote):

```
(defun cylinder-volume (radius height)
  (let (area)
    (setq area (* pi (expt radius 2)))
    (* area height) ) )
```

List Structure

cons: 

Lists are a basic data structure in Lisp; in fact, Lisp code is made of lists. The external (printed) representation of lists is a sequence of elements enclosed in parentheses.

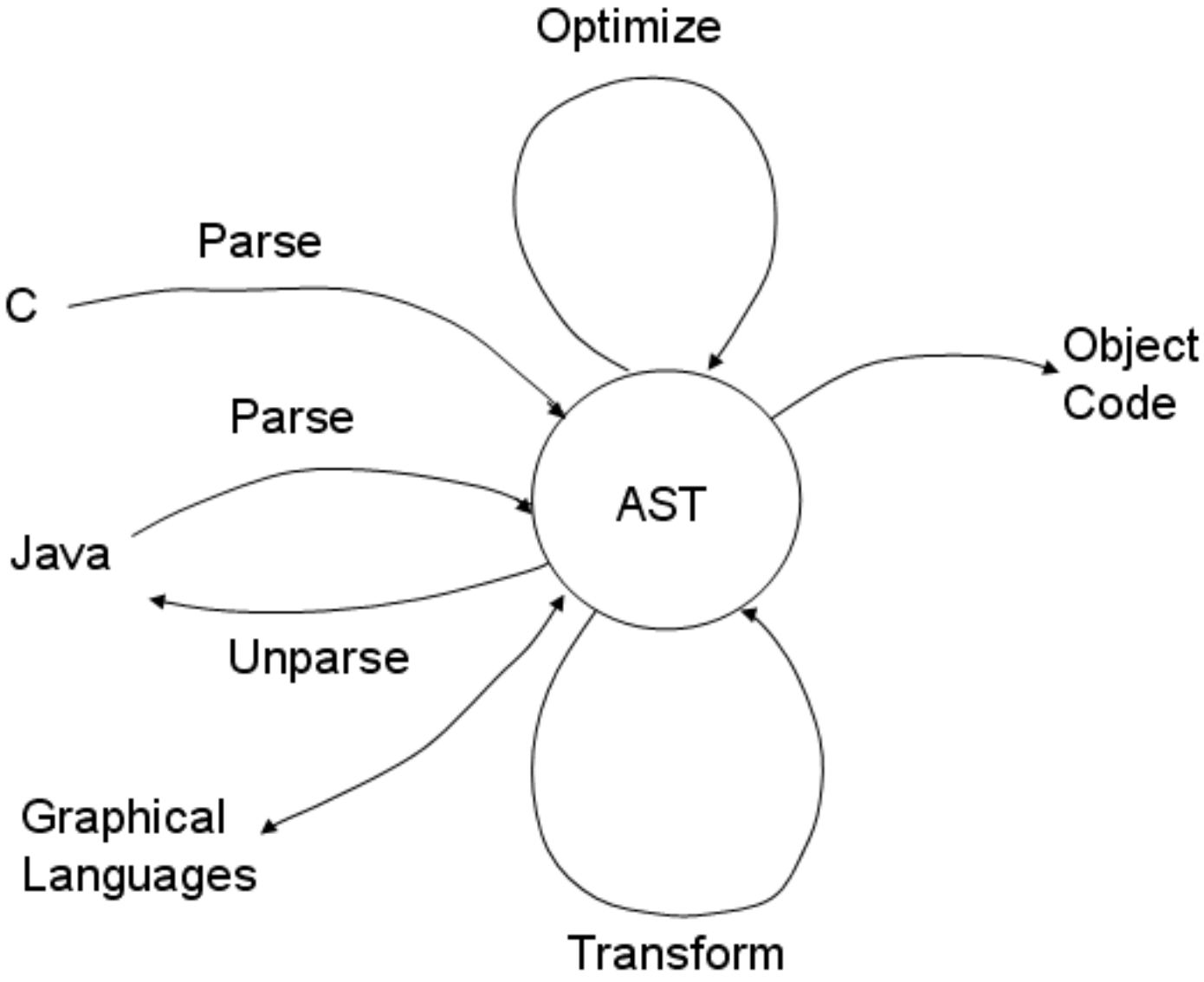
```
(first '(a b c))        =  A

(rest '(a b c))         =  (B C)

(second '(a b c))       =  B

(cons 'new '(a b c))    =  (NEW A B C)

(list 'a 'b 'c)         = (A B C)
```

`first` is also called `car`; `rest` is also called `cdr`.

The quote symbol `'` is a shorthand for the pseudo-function `quote`. `(quote x) = x`, that is, quote returns the argument itself rather than evaluating the argument.

Abstract Syntax Tree

We consider the fundamental form of a program to be the *abstract syntax tree* (AST) – not source code.



Lisp code is already in AST form, and Lisp is ideal for implementing program generation and transformation. It is easy to generate code in ordinary programming languages from Lisp.

Binding Lists

A *binding* is a correspondence of a name and a value.

A set of bindings is represented as a list, called an *association list*, or *alist* for short. A new binding can be added by:
`(push (list` *name value* `)` *binding-list* `)`

A name can be looked up using `assoc`:
`(assoc` *name binding-list* `)`

```
(assoc '?y '((?x 3) (?y 4) (?z 5)))
   =  (?Y 4)
```

The value of the binding can be gotten using `second`:

```
(second (assoc '?y '((?x 3) (?y 4) (?z 5))))
   =  4
```

Substitution

`(subst x y z)` ("substitute x for y in z") can be used to make new code from a pattern.

```
>(subst pi 'pi '(* pi (* r r)))
(* 3.14159265 (* R R))

>(subst 1 'i '(aref x (+ -8 (* 8 i))))
(AREF X (+ -8 (* 8 1)))
```

`(sublis alist form)` makes multiple substitutions:

```
>(sublis '((rose . peach) (smell . taste))
         '(a rose by any other name
           would smell as sweet))
(A PEACH BY ANY OTHER NAME WOULD TASTE AS SWEET)
```

Copying and Substitution Functions [48]

```
(defun copy-tree (z)
  (if (consp z)
      (cons (copy-tree (first z))
            (copy-tree (rest z)))
      z) )

; substitute x for y in z
(defun subst (x y z)
  (if (consp z)
      (cons (subst x y (first z))
            (subst x y (rest z)))
      (if (eql z y) x z)) )

; substitute in z with bindings in alist
(defun sublis (alist z)
  (let (pair)
    (if (consp z)
        (cons (sublis alist (first z))
              (sublis alist (rest z)))
        (if (setq pair (assoc z alist))
            (cdr pair)
            z)) ))
```

[48] These are system functions in Common Lisp. The system functions `subst` and `sublis` copy only as much structure as necessary.

Substitution in C

```
/* Substitute new for old in tree */
TOKEN subst (TOKEN new, TOKEN old, TOKEN tree)
{  TOKEN tok, last, opnd, ptr;
   if (tree == NULL) return (tree);
   if (tree->tokentype == OPERATOR)
      {  last = NULL;
         ptr = tree->operands;
         tok = copytok(tree);
         while ( ptr != NULL )
            {  opnd = subst (new, old, ptr);
               if (last == NULL)
                        tok->operands = opnd;
                  else last->link = opnd;
               last = opnd;
               ptr = ptr->link;
            }
         return (tok) ;
        }
     else if (tree->tokentype == IDENTIFIERTOK
              && strcmp(tree->stringval,
                        old->stringval) == 0)
            return ( copytok(new) );
        else return ( copytok(tree) );
}
```

Loop Unrolling

Substitution makes it easy to do loop unrolling:

```
(defun unroll (var n code)
  (let (res)
    (dotimes (i n)
      (push (subst (1+ i) var code) res))
    (cons 'progn (reverse res)) ))


>(unroll 'j 5 '(|:=| (aref x (+ -8 (* 8 j))) 0))

(PROGN
  (|:=| (AREF X (+ -8 (* 8 1))) 0)
  (|:=| (AREF X (+ -8 (* 8 2))) 0)
  (|:=| (AREF X (+ -8 (* 8 3))) 0)
  (|:=| (AREF X (+ -8 (* 8 4))) 0)
  (|:=| (AREF X (+ -8 (* 8 5))) 0))
```

Instantiating Design Patterns

`sublis` can instantiate design patterns. For example, we can instantiate a tree-recursive accumulator pattern to make various functions:

```
(setq pattern
  '(defun ?fun (tree)
     (if (consp tree)
         (?combine (?fun (car tree))
                   (?fun (cdr tree)))
         (if (?test tree) ?trueval ?falseval))))

>(sublis '((?fun . nnums)
           (?combine . +)
           (?test . numberp)
           (?trueval . 1)
           (?falseval . 0))      pattern)

(DEFUN NNUMS (TREE)
  (IF (CONSP TREE)
      (+ (NNUMS (CAR TREE))
         (NNUMS (CDR TREE)))
      (IF (NUMBERP TREE) 1 0)))

>(nnums '(+ 3 (* i 5)))
2
```

Pattern Matching

Pattern matching is the inverse of substitution: it tests to see whether an input is an instance of a pattern, and if so, how it matches. [49]

```
(match '(defun ?fun (tree)
          (if (consp tree)
              (?combine (?fun (car tree))
                        (?fun (cdr tree)))
              (if (?test tree) ?trueval ?falseval))
       '(DEFUN NNUMS (TREE)
          (IF (CONSP TREE)
              (+ (NNUMS (CAR TREE))
                 (NNUMS (CDR TREE)))
              (IF (NUMBERP TREE) 1 0))) )

((?FALSEVAL . 0) (?TRUEVAL . 1) (?TEST . NUMBERP)
 (?COMBINE . +) (?FUN . NNUMS) (T . T))

(match '(- ?x (- ?y))
       '(- z (- (* u v))))

((?Y * U V) (?X . Z) (T . T))
```

[49] The pattern matcher code can be loaded using `(load "/projects/cs375/patmatch.lsp")` .

Pattern Matching

```
(defun equal (pat inp)
  (if (consp pat)           ; interior node?
      (and (consp inp)
           (equal (car pat) (car inp))
           (equal (cdr pat) (cdr inp)))
      (eql pat inp) ) )    ; leaf node

(defun match (pat inp) (matchb pat inp '((t . t))))
(defun matchb (pat inp bindings)
  (and bindings
    (if (consp pat)           ; interior node?
        (and (consp inp)
             (matchb (cdr pat)
                     (cdr inp)
                     (matchb (car pat)
                             (car inp) bindings)))
        (if (varp pat)        ; leaf: variable?
            (let ((binding (assoc pat bindings)))
              (if binding
                  (and (equal inp (cdr binding))
                       bindings)
                  (cons (cons pat inp) bindings)))
            (and (eql pat inp) bindings)) ) ) )
```

Transformation by Patterns

Matching and substitution can be combined to transform an input from a `pattern-pair`: a list of input pattern and output pattern.

```
(defun transform (pattern-pair input)
  (let (bindings)
    (if (setq bindings
              (match (first pattern-pair) input))
        (sublis bindings (second pattern-pair))) ))

>(transform '( (- ?x (- ?y))  (+ ?x ?y) )
            '(- z (- (* u v))) )

(+ Z (* U V))


>(transform '((- (+ ?x ?y) (+ ?z ?y))
              (- ?x ?z))
            '(- (+ (age tom) (age mary))
                (+ (age bill) (age mary))))

(- (AGE TOM) (AGE BILL))
```

Transformation Patterns

Optimization:

```
(defpatterns 'opt
  '( ((+ ?x 0)              ?x)
     ((* ?x 0)               0)
     ((* ?x 1)              ?x)
     ((:= ?x (+ ?x 1))     (incf ?x)) ))
```

Language translation:

```
(defpatterns 'lisptoc
  '( ((aref ?x ?y)        ("" ?x "[" ?y "]"))
     ((incf ?x)           ("++" ?x))
     ((+ ?x ?y)           ("(" ?x " + " ?y ")"))
     ((= ?x ?y)           ("(" ?x " == " ?y ")"))
     ((and ?x ?y)         ("(" ?x " && " ?y ")"))
     ((if ?c ?s1 ?s2)     ("if (" ?c ")" #\Tab
                            #\Return ?s1
                            #\Return ?s2))
```

Program Transformation using Lisp

```
>code
(IF (AND (= J 7) (<> K 3))
    (PROGN (:= X (+ (AREF A I) 3))
           (:= I (+ I 1))))

>(cpr (transform (transform code 'opt)
                 'lisptoc))

if (((j == 7) && (k != 3)))
  {
    x = (a[i] + 3);
    ++i;
    }
```

Dot Matching

It is possible to use "dot notation" to match a variable to the rest of a list:

```
( (progn nil . ?s)   (progn . ?s) )
```

The variable `?s` will match whatever is at the end of the list: 0 or more statements.

```
(transf '( (progn nil . ?s)   (progn . ?s) )
        '(progn nil (setq x 3) (setq y 7)) )

(PROGN (SETQ X 3) (SETQ Y 7))
```

Looping Patterns

```
((for ?i ?start ?end ?s)

 (PROGN (\:= ?i ?start)
        (LABEL ?j)
        (IF (<= ?i ?end)
            (PROGN ?s
                   (\:= ?i (+ 1 ?i))
                   (GOTO ?j) )))

 t ((?j (gentemp "LABEL"))) )

((while ?c ?s)

 (PROGN (LABEL ?j)
        (IF ?c (PROGN ?s (GOTO ?j))))

 t ((?j (gentemp "LABEL"))) )

((repeat-until ?c . ?s)

 (PROGN (LABEL ?j)
        (progn . ?s)
        (IF ?c (PROGN) (GOTO ?j)))
 t ((?j (gentemp "LABEL"))) )
```

Code Expansion by Looping Patterns

```
>(trans '(for i 1 100
           (\:= sum (+ sum (aref x (* 8 i)))))
        'loops)

(PROGN
  (|:=| I 1)
  (LABEL LABEL7)
  (IF (<= I 100)
      (PROGN
        (|:=| SUM (+ SUM (AREF X (* 8 I))))
        (|:=| I (+ 1 I))
        (GOTO LABEL7))))

>(trans '(repeat-until (> i 100)
           (writeln i) (\:= i (+ i 1)))
        'loops)

(PROGN
  (LABEL LABEL8)
  (PROGN (WRITELN I) (|:=| I (+ I 1)))
  (IF (> I 100) (PROGN) (GOTO LABEL8)))
```

More Complex Rules

It is desirable to augment rewrite rules in two ways:

1. Add a predicate to perform tests on the input; only perform the transformation if the test succeeds:
   `(and (numberp ?n) (> ?n 0))`
2. Create new variables by running a program on existing variables:

```
(transf '((intersection
           (subset (function (lambda (?x) ?p))
                   ?s)
           (subset (function (lambda (?y) ?q))
                   ?s))
          (subset (function (lambda (?x)
                     (and ?p ?qq)))
                  ?s)
          t
          ((?qq (subst ?x ?y ?q))) )
  '(intersection
     (subset #'(lambda (w) (rich w)) people)
     (subset #'(lambda (z) (famous z)) people)))

(SUBSET #'(LAMBDA (W) (AND (RICH W) (FAMOUS W)))
        PEOPLE))
```

Multi-Level Patterns

```
(redefpatterns 'loop
     ((average ?set)
       (make-loop ?set ?item (?total ?n)
        (progn (setq ?total 0)
               (setq ?n 0))
        (progn (incf ?total ?item)
               (incf ?n))
               (/ ?total ?n) )
        t ((?item (gentemp "ITEM"))
           (?total (gentemp "TOTAL"))
           (?n (gentemp "N"))) ) ) )

(redefpatterns 'list
  '( ((make-loop ?lst ?item ?vars ?init ?action ?re
        (let (?ptr ?item . ?vars)
          ?init
          (setq ?ptr ?lst)
          (while ?ptr
            (setq ?item (first ?ptr))
            (setq ?ptr (rest ?ptr))
            ?action)
          ?result)
        t ((?ptr (gentemp "PTR"))) ) ) )
```

Use of Multi-Level Patterns

```
(cpr (trans
      (trans
       (trans '(defun zb (x) (average x))
              'loop) 'list) 'lisptoc))

zb(x)
  {
    int ptr30; int item27; int total28; int n29;;
    {
      total28 = 0;
      n29 = 0;
      };
    ptr30 = x;
    while ( ptr30 )
      {
        item27 = first(ptr30);
        ptr30 = rest(ptr30);
        {
          total28 += item27;
          ++n29;
          };
        };
    return ((total28 / n29));
    };
```

Function Inlining

Inlining is the expansion of the code of a function at the point of call. If the code says `sqrt(x)`, `sqrt` can be invoked as a *closed* function in the usual way, or it can be expanded as an *open* or *inline* function by expanding the definition of `sqrt` at each point of call.

Inline expansion saves the overhead of subroutine call and parameter transmission; it may allow additional optimization because the compiler can now see that certain things (including types) are constant.

If code is in the form of abstract syntax trees, inlining is easy:

- Make sure the variables of the function are distinct from those of the caller.
- Generate assignment statements for the arguments.
- Copy the code of the function.

Program Transformation

Many kinds of transformations of a program are possible:

- Optimization of various kinds. Low-level inefficiencies created by a program generation system can be removed.
- Specialization. Generic operations or program patterns can be specialized to the form needed for a specific implementation of an abstract data structure. OOP methods can be specialized for subclasses.
- Language translation. Transformations can change code into the syntax of the target language.
- Code expansion. Small amounts of input code can be transformed into large amounts of output code. The expansion can depend on specifications that are much smaller than the final code.
- Partial evaluation. Things that are constant at compile time can be evaluated and eliminated from code.
- Changing recursion to iteration
- Making code more readable
- Making code *less* readable (*code obfuscation*)

Pattern Optimization Examples

```
(defun t1 (C D)
  (COND ((> (* PI (EXPT (CADDR (PROG1 C)) 2))
            (* PI (EXPT (CADDR (PROG1 D)) 2)))
         (PRINT 'BIGGER))))

(LAMBDA-BLOCK T1 (C D)
  (IF (> (ABS (CADDR C)) (ABS (CADDR D)))
         (PRINT 'BIGGER)))
```

Examples ...

```
(defun t2 (P Q)
  (LET ((DX (- (- (+ (CADDR (CURRENTDATE)) 1900)
                  (+ (CADDR (GET (PROG1 P)
                                 'BIRTHDATE))
                     1900))
               (- (+ (CADDR (CURRENTDATE)) 1900)
                  (+ (CADDR (GET (PROG1 Q)
                                 'BIRTHDATE))
                     1900))))
        (DY (- (/ (GET (PROG1 P) 'SALARY) 1000.0)
               (/ (GET (PROG1 Q) 'SALARY)
                  1000.0))))
       (SQRT (+ (* DX DX) (* DY DY)))))

(LAMBDA-BLOCK T2 (P Q)
  (LET ((DX (- (CADDR (GET Q 'BIRTHDATE))
               (CADDR (GET P 'BIRTHDATE))))
        (DY (/ (- (GET P 'SALARY)
                  (GET Q 'SALARY))
               1000.0)))
    (SQRT (+ (* DX DX) (* DY DY)))))
```

Examples ...

```
(defun t3 (P)
  (> (* PI (EXPT (/ (CADDR (PROG1 P)) 2) 2))
     (* (- (* PI (EXPT (/ (CADDR (PROG1 P)) 2) 2))
           (* PI (EXPT (/ (CADR (PROG1 P)) 2) 2)))
        (GET (FIFTH (PROG1 P)) 'DENSITY))))

(LAMBDA-BLOCK T3 (P)
  (> (EXPT (CADDR P) 2)
     (* (- (EXPT (CADDR P) 2) (EXPT (CADR P) 2))
        (GET (FIFTH P) 'DENSITY))))


(defun t4 ()
  (cond ((> 1 3) 'amazing)
        ((< (sqrt 7.2) 2) 'incredible)
        ((= (+ 2 2) 4) 'okay)
        (t 'jeez)))

(LAMBDA-BLOCK T4 () 'OKAY)
```

Examples ...

```
(defun t5 (C)
  (DOLIST
    (S (INTERSECTION
         (SUBSET #'(LAMBDA (GLVAR7289)
                     (EQ (GET (PROG1 GLVAR7289)
                              'SEX)
                         'FEMALE))
                 (GET (PROG1 C) 'STUDENTS))
         (SUBSET #'(LAMBDA (GLVAR7290)
                     (>= (STUDENT-AVERAGE
                          (PROG1 GLVAR7290))
                         95))
                 (GET (PROG1 C) 'STUDENTS))))
    (FORMAT T "~A ~A~%" (GET S 'NAME)
            (STUDENT-AVERAGE S))))

(LAMBDA-BLOCK T5 (C)
  (DOLIST (S (GET C 'STUDENTS))
    (IF (AND (EQ (GET S 'SEX) 'FEMALE)
             (>= (STUDENT-AVERAGE S) 95))
        (FORMAT T "~A ~A~%" (GET S 'NAME)
                (STUDENT-AVERAGE S)))))
```

Paul Graham:

"If you ever do find yourself working for a startup, here's a handy tip for evaluating competitors. Read their job listings...

After a couple years of this I could tell which companies to worry about and which not to. The more of an IT flavor the job descriptions had, the less dangerous the company was. The safest kind were the ones that wanted Oracle experience. You never had to worry about those. You were also safe if they said they wanted C++ or Java developers. If they wanted Perl or Python programmers, that would be a bit frightening – that's starting to sound like a company where the technical side, at least, is run by real hackers. If I had ever seen a job posting looking for Lisp hackers, I would have been really worried."

English

English is a context-free language (more or less).

English has a great deal of ambiguity, compared to programming languages. By restricting the language to an *English subset* for a particular application domain, English I/O can be made quite tractable.

Some users may prefer an English-like interface to a more formal language.

Of course, the best way to process English is in Lisp.

Expression Trees to English [50]

```
(defun op  (x) (first x))
(defun lhs (x) (second x))
(defun rhs (x) (third x))

(defun op->english (op)
  (list 'the
      (second (assoc op '((+ sum)
                          (- difference)
                          (* product)
                          (/ quotient)
                          (sin sine)
                          (cos cosine)))) 'of))

(defun exp->english (x)
  (if (consp x)                    ; operator?
      (append
        (op->english (op x))
        (exp->english (lhs x))
        (if (null (cddr x))        ; unary?
            '()
            (cons 'and
                  (exp->english (rhs x)) ) ) )
      (list x) ) )                 ; leaf: operand
```

[50] file `expenglish.lsp`

Generating English

```
%lisp
>(load "/projects/cs375/expenglish.lsp")

>(exp->english 'x)

(X)

>(exp->english '(+ x y))

(THE SUM OF X AND Y)

>(exp->english '(/ (cos z) (+ x (sin y))))

(THE QUOTIENT OF THE COSINE OF Z AND
     THE SUM OF X AND THE SINE OF Y)
```

Parsing English

In most cases, a parser for a programming language never has to back up: if it sees `if`, the input must be an `if` statement or an error.

Parsing English requires that the parser be able to fail, back up, and try something else: if it sees `in`, the input might be `in Austin` or `in April`, which may be handled by different kinds of grammar rules.

Backup means that parsing is a search process, i.e. likely to be NP-complete. However, since English sentences are usually short, this is not a problem in practice.

An *Augmented Transition Network* (ATN) framework facilitates parsing of English.

ATN in Lisp [51]

- A global variable `*sent*` points to a list of words that is the remaining input sentence:
  `(GOOD CHINESE RESTAURANT IN LOS ALTOS)`
- A global variable `*word*` points to the current word: `GOOD`
- `(cat` *category* `)` tests whether a word is in the specified category. It can also translate the word, e.g. `(cat 'month)` might return `3` if `*word*` is `MARCH`.
- `(next)` moves to the next word in the input
- `(saveptr)` saves the current sentence position on a stack.
- `(success)` pops a saved position off the stack and returns `T`.
- `(fail)` restores a saved position from the stack (restoring `*sent*` and `*word*`) and returns `NIL`.

[51] file `atn.lsp`

Parsing Functions

The parser works by recursive descent, but with the ability to fail and back up and try another path.

```
(defun loc ()
 (let (locname)
  (saveptr)
  (if (and (eq *word* 'in) (next)
           (setq locname (cat 'city))
           (next))
      (progn
        (addrestrict
          (list 'equal
                (dbaccess 'customer-city)
                (kwote locname)))
        (success))
      (fail) ) ))
```

Grammar Compiler [52]

It is easy to write a grammar compiler that converts a Yacc-like grammar into the equivalent ATN parsing functions. This is especially easy in Lisp since Lisp code and Lisp data are the same thing.

```
(rulecom '(LOC  -> (in (city))
               (restrict 'customer-city $2)) )

(DEFUN LOC62 ()
  (LET ($1 $2)
    (SAVEPTR)
    (IF (AND (AND (EQL (SETQ $1 *WORD*) 'IN)
                  (NEXT))
             (SETQ $2 (CITY)))
        (PROGN (SUCCESS)
               (RESTRICT 'CUSTOMER-CITY $2))
        (FAIL))))
```

[52] file `gramcom.lsp`

Access to Database[53]

English can be a good language to use query a database.

```
(deflexicon
 '((a/an     (a an some))
   (i/you    (i you one))
   (get      (get find obtain))
   (quality  ((good 2.5) ))
   (restword (restaurant
              (restaurants restaurant)))
   ))
```

[53] file `restgram.lsp`

Restaurant Database Grammar

```
(defgrammar
  (s -> ((command) (a/an)? (qual)? (resttype)?
         (restword) (qualb)? (loc)?)
        (makequery (combine (retrieve 'restaurant)
                            (retrieve 'streetno)
                            (retrieve 'street)
                            (retrieve 'rating)
                            $3 $4 $6 $7)))
  (s -> (where can (i/you) (get) (qual)?
                (resttype)? food ? (loc)?)
        (makequery (combine (retrieve 'restaurant)
                            (retrieve 'streetno)
                            (retrieve 'street)
                            (retrieve 'rating)
                            $5 $6 $8)))
  (command  ->  (what is) t)
  (qual     ->  ((quality))
                      (restrictb '>= 'rating $1))
  (qualb    ->  (rated above (number))
                      (restrictb '>= 'rating $3))
  (resttype ->  ((kindfood))
                      (restrict 'foodtype $1))
  (loc      ->  (in (city))  (restrict 'city $2))
```

Restaurant Queries

```
%lisp
>(load "/projects/cs375/restaurant.lsp")

>(askr '(where can i get ice cream in berkeley))

((2001-FLAVORS-ICE-CREAM-&-YOGUR 2485 TELEGRAPH-AVE
 (BASKIN-ROBBINS 1471 SHATTUCK-AVE)
 (DOUBLE-RAINBOW 2236 SHATTUCK-AVE)
 (FOSTERS-FREEZE 1199 UNIVERSITY-AVE)
 (MARBLE-TWENTY-ONE-ICE-CREAM 2270 SHATTUCK-AVE)
 (SACRAMENTO-ICE-CREAM-SHOP 2448 SACRAMENTO-ST)
 (THE-LATEST-SCOOP 1017 ASHBY-AVE))

>(askr '(show me chinese restaurants
         rated above 2.5 in los altos))

((CHINA-VALLEY 355 STATE-ST)
 (GRAND-CHINA-RESTAURANT 5100 EL-CAMINO-REAL)
 (HUNAN-HOMES-RESTAURANT 4880 EL-CAMINO-REAL)
 (LUCKY-CHINESE-RESTAURANT 140 STATE-ST)
 (MANDARIN-CLASSIC 397 MAIN-ST)
 (ROYAL-PALACE 4320 EL-CAMINO-REAL))
```

Physics Problems[54]

```
(deflexicon
 '((propname (radius diameter circumference
      area volume height velocity time
      weight power height work speed mass))
   (a/an     (a an))
   (the/its  (the its))
   (objname  (circle sphere fall lift))
)) ; deflexicon

(defgrammar
  (s         -> (what is (property) of (object))
                (list 'calculate $3 $5))
  (property  -> ((the/its)? (propname)) $2)
  (quantity  -> ((number)) $1)
  (object    -> ((a/an)? (objname) with (objprops))
                (cons 'object (cons $2 $4)))
  (objprops  -> ((objprop) and (objprops))
                (cons $1 $3))
  (objprops  -> ((objprop))  (list $1))
  (objprop   -> ((a/an)? (propname) of ? (quantity)
                (cons $2 $4))
  (objprop   -> ((propname) = (quantity))
                (cons $1 $3))                    )
```

[54] file `physgram.lsp`

Physics Queries

```
%lisp
>(load "/projects/cs375/physics.lsp")

>(phys '(what is the area of a circle
              with diameter = 10))

78.539816339744831


>(phys '(what is the circumference of a circle
              with an area of 100))

35.449077018110316


>(phys '(what is the power of a lift with
              mass = 100 and height = 6
              and time = 10))

588.399
```