

Compilers for Algorithmic Languages
Fifth Edition

University of Kentucky CS 441G
Fall, 2006

Craig C. Douglas

University of Kentucky
Department of Computer Science
771 Anderson Hall
Lexington, KY 40506-0046, USA

Craig.Douglas@uky.edu
<http://www.mgnet.org/~douglas>

MGNet.org
Cos Cob, CT, USA

Copyright © 2006
All rights reserved

Modules of a Compiler

Actual compiler:

1. What is the
 - a. Language
 - b. Target output:
 - i. Simple use of an upper level language (e.g., C), i.e., a substitute for assembler
 - ii. Complicated use of an upper level language, i.e., a simple translator
 - iii. Assembly code for specific processor and assembler
 - iv. XML
 - v. Executable machine code (load and go)
2. Lexical analysis
3. Symbol table
4. Parsing
5. Errors
6. Code generation
7. Optimization
 - a. Local
 - b. Global

Post compiler:

1. Assembly/convert to machine language
2. Loader/linker/load and go
3. Run the code

What is the Language

Let us look at an example and try to figure out what it actually ought to be. It is available from the notes web page.

mg.f06:

```
// -----  
//  
// This is the Fall 2006 target code for CS 441G.  
//  
// Your compiler should be able to lex, parse, and generate code for the  
// entire file by the end of the course.  
//  
// Written by: Craig C. Douglas  
// Modification history:  
//     Thu Aug 17 15:34:01 EDT 2006     First take on project file  
//  
// -----  
  
// -----  
//  
// You can find a lot of information about multigrid methods at the web site  
//  
//     http://www.mgnet.org  
//  
// Click on Tutorials after reading why it is so great to work in this field.  
//  
// -----  
//  
// ***** Watch out for inconsistencies in this file and report them. *****  
//  
// -----  
  
program $two_grid_solver  
{  
  
    // -----  
    // Place a constant in every element of a vector.  
    // -----  
  
    procedure  
        set_constant(  
            double dval,                // The constant value  
            double dsoln[], integer s1  // Approximate solution  
        )  
    {  
        integer i := 0;                // Loop variable  
  
        while ( i >= 0 && i <= s1 )  
            dsoln[i++] := dval;
```

```

} // of set_constant

// -----
// Print every element of a vector, one per line with the index.
// -----

procedure
  print_vector(
    string title,           // Identification of vector
    double dsoln[], integer s1 // Approximate solution
  )
{
  integer i;

  print_string( "Vector: " );
  print_string( "title" );
  print_string( "\ni value\n" );
  do ( i := 0; i <= s1; i++ )
  {
    print_integer( i );
    print_string( " " );
    print_double( dsoln[i] );
  }
  print_string( "--- End of vector\n" );
} // of print_vector

// -----
// Calculate the little ell-infinity norm of the error in the solution.
// -----

function
double error_norm(
  double dsoln[], integer s1 // Approximate solution
)
{
  integer i := 0; // Loop variable
  double asoln; // abs(dsoln[i])
  double l0_norm := 0.0d0; // Little L1 norm

  // The real solution is uniformly 0, so the maximum error is the
  // absolute value of the approximate solution

  while ( i <= s1 )
  {
    if ( dsoln[i] <= 0. ) then
      asoln := -dsoln[i];
    else
      asoln := dsoln[i];
    if ( asoln > l0_norm ) then
    {
      l0_norm := asoln;
    }
    i++;
  }

  return l0_norm;
} // of error_norm

```

```

// -----
// Compute the residual vector.
// -----

procedure
  residuals(
    double  dsoln[], integer s1,          // Approximate solution
    double  drhs[], integer rhs1,       // Right hand side
    double  dres[], integer res1        // Residuals
  )
{
  integer i;                            // Loop variable

  // Compute the residuals
  dres[0] := dres[res1] := 0.0d0;
  do ( i := 1; i < s1; i++ )
    dres[i] := drhs[i] - 2.0 * dsoln[i]
              + dsoln[i-1]
              + dsoln[i+1];

} // of residuals

// -----
// Do some Gauss-Seidel iterations to approximate the solution.
// -----

function
double gauss_seidel(
  integer iters,                          // Number of iterations
  double dsoln[], integer s1,             // Approximate solution
  double drhs[], integer rhs1            // Right hand side
)
{
  integer i, n=1;                          // Loop variables

  // Do iters number of Gauss-Seidel iterations
  while (n<=iters)
  {
    do ( i := 1; i < s1; i++ )
      dsoln[i] := ( drhs[i] + dsoln[i-1]
                    + dsoln[i+1]) / 2.0d0;

    n++;
  }

  // Return the error norm
  return error_norm( dsoln, s1 );

} // of gauss_seidel

// -----
// Interpolate between the two grids.
// -----

function
integer interpolate(
  double dfrom[], integer f1,            // Original data, sized (f1)
  double dto[], integer t1               // Target date, sized (t1)
)
{
  // Two procedures defined only inside of interpolate

```

```

// -----
// Interpolate from the finer mesh to the coarser mesh.
// -----

procedure
  coarsen(
    double dfrom[], integer f1,      // Original data, sized (f1)
    double dto[], integer t1       // Target date, sized (t1)
  )
{
  integer i, m;          // Loop variables

  // Aggregate the from data in a Galerkin style on the coarser mesh
  dto[0] := dto[t1] := 0.;
  m := 0;
  do ( i := 1 ; i < t1 ; i++ )
  {
    m += 2;
    dto[i] := dfrom[m] +
              5.d-1 * ( dfrom[m-1] + dfrom[m+1] );
  }
} // of coarsen

// -----
// Interpolate from the coarser mesh to the finer mesh and add to an
// already existing approximate solution.
// -----

procedure
  refine_add(
    double dfrom[], integer f1,      // Original data, sized (f1)
    double dto[], integer t1       // Target date, sized (t1)
  )
{
  integer i, m;          // Loop variables

  // Deal with mesh points coincident between the two meshes
  m := 0;
  do ( i := 1; i < f1 ; i++ )
  {
    m := m + 2;
    dto[m] := dto[m] + dfrom[i];
  }

  // Deal with mesh points noncoincident between the two meshes
  m := -1;
  do ( i := 0; i < f1; i++ )
  {
    m := m + 2;
    dto[m] := dto[m] +
              .5 * ( dfrom[i] + dfrom[i+1] );
  }
} // of refine_add

// interpolate's code really starts here

// Interpolate to a coarser mesh
if ( t1 == f1 / 2 ) then

```

```

        coarsen( dfrom, f1, dto, t1 );

// Interpolate and add to what is on a finer mesh
else if ( t1 == f1 * 2 ) then
{
    refine_add( dfrom, f1, dto, t1 );
}

// Uh, oh... this is incompatible
else
{
    print_string( "Error in routine interp: data size mismatch.\n" );
    return 0;
}
return 1;
} // of interpolate

// -----
// The actual two grid multilevel algorithm.
// -----

function
integer main(
    )
{
    integer rval := 0;           // Return value
    integer fm1=1, cm1;        // Fine and coarse mesh upper limits
    double  enorm;             // Error norm

    // Determine fine mesh size. Coarse mesh is roughly half the size.
    while( fm1 <= 4 || fm1 % 2 != 0 )
    {
        print_string( "Number of points in the fine mesh (must be even and
at least 6) " ); // Word wrapped in Word document: all on one line in file
        read_integer( fm1 );
    }
    cm1 := fm1 / 2;
    print_string( "Fine  mesh points 0:" );
    print_integer( fm1 );
    print_string( "\nCoarse mesh points 0:" );
    print_integer( cm1 );
    print_string( "\n" );

    // Allocate space dynamically
    double fm[fm1+1],          // Fine grid approximate solution
           frhs[fm1+1],       // Fine grid right hand side
           fres[fm1+1];       // Fine grid residuals
    double cm[cm1+1], crhs[cm1+1]; // Coarse grid solution and right
                                   // hand side

    // Set the initial guess to the solution
    set_constant( 1.0d0, fm, fm1 );
    fm[0] := 0.0d0;
    fm[fm1] := 0.;
    print_vector( "Initial guess", fm, fm1 );

    // Get the initial error norm
    enorm := error_norm( fm, fm1 );
    print_string( "initial error norm := " );
    print_double( enorm );
    print_string( "\n" );

```

```

// Do some Gauss-Seidel iterations on the fine mesh
enorm := gauss_seidel( 4, fm, fml, frhs, fml );
print_vector( "after first fine mesh smoothing", fm, fml );
print_string( "Fine mesh error norm := " );
print_double( enorm );
print_string( "\n" );

// Compute the residuals on the fine mesh and project them onto the
// coarse mesh right hand side.
residuals( fm, fml, frhs, fml, fres, fml );
print_vector( "Residuals on fine mesh", fres, fml );
if ( interpolate( fres, fml, crhs, cml ) != 0 ) then
    return rval := 1;

// Do some Gauss-Seidel iterations on the coarse mesh
enorm := gauss_seidel( 500, cm, cml, crhs, cml );
print_vector( "coarse mesh correction", cm, cml );

// Interpolate the correction to the fine grid
if ( interpolate( cm, cml, fm, fml ) > 0 ) then
    return 2;
enorm := error_norm( fm, fml );
print_string( "Fine mesh error norm := " );
print_double( enorm );
print_string( "\n" );
print_vector( "after interpolation to fine mesh", fm, fml );

// Do some Gauss-Seidel iterations on the fine mesh
enorm := gauss_seidel( 4, fm, fml, frhs, fml );
print_vector( "after second fine mesh smoothing", fm, fml );
print_string( "Fine mesh error norm := " );
print_double( enorm );
print_string( "\n" );

// All done. Return 0 if everything worked out or something else if
// something went wrong.
return rval;

} // of main

} // of program $two_grid_solver

```

What is the Target?

You will generate very simple C code that you will pass through gcc. To be safe, you should check that your code works on a Mac OS X system. (e.g., one of the CS Lab machines). First, from the notes web page, get the file

f06.c

Note that it defines a number of things, including the number of registers and size of memory. It has its own main program, which includes your generated routine by including the file *yourmain.h*. The main program in f06.c calls your main program using the following statement:

`yourmain()`

Do not modify the file f06.c since I will use the class definition, not yours.

Should you modify the class definition, you had better be able to justify the modification so that the entire class uses it, too.

Note that your include file is independent of what language your compiler is written in since all of the code to be compiled will be turned into pigeon C.

What is in f06.c?

- A bunch of #define's.
- The global definitions for the memory types.
- A collection of useful functions and procedures that your generated code will call similar to system calls in a C program.
- A main program that initializes the stack register, calls yourmain(), and then returns.
- A routine, F06_Exit, that prints statistics and exits.

When in doubt, look at the current version of the file. If something is odd, ask for an explanation.

Warning: The file will for sure change during the course based on requests for extra functionality and the whims of the professor. It is a good idea to download it often and certainly after email announcements about changes.

Memory and Registers

The memory is broken up into several objects:

- Integer Registers $R[i]$, $0 \leq i < F06_RSize$
- Floating Point Registers $F[i]$, $0 \leq i < F06_FSize$
- One Stack Register SR (an index into Mem)
- One Frame Register FR (an index into Mem)
- One Main Memory $Mem[i]$,
 $0 \leq i < F06_MemSize$
- Aliases to Mem called FMem and SMem for floating point and character strings
- One Timing Register F06_Time
- One String Buffer area F06_SBuf (length 1024)

Note that R's are always *integer* valued and F's are always *double* valued. Mem is also *integer* valued, which makes addressing awkward for anything else e.g., doubles or strings. Hence, the aliases FMem and SMem have been provided to ease code generation.

You will reference these objects in a simple manner. The result of all operations will be a particular register, e.g., $R[3]$ or $F[2]$ (i.e., not $R[i]$).

The stack register SR will initially be set to $F06_MemSize-1$, i.e., the end of memory. *The stack will grow down from the end of memory. The dynamic memory heap grows up from the beginning of memory.*

The frame register FR is initially set to the same value as the stack register. You may manipulate FR anyway you see fit. It is an integer pointer, however, and should only point into Mem.

You will dynamically allocate memory from the beginning of memory. Memory routines are part of f06.c. You allocate and free memory with

- *int allocate_in_Mem(int nints)*
which allocates nints words in Mem. The index returned is always an even number so that you can easily allocate floating point numbers.
- *void free_in_Mem(int ptr)*
which frees the space allocated by the index into Mem, *ptr*, returned by *allocate_in_Mem* in an earlier call.

Lexical Analysis

Recognizes the “words” of a programming language.
Typically,

1. *Names for variables*: usually a letter followed by letters, digits, or underscores.
2. *Operators*: +, -, //, :=, !, ;
3. *Key words*: if, allocate, function
4. *Numbers*:
 - a. Integers: strings of digits
 - b. Reals: Fraction + Exponent
 - c. Complex: two reals
5. *Character vectors* or *strings*: surrounded by quotes
6. *Space*: usually used as a delimiter and usually just thrown away.

Sometimes *constants* are defined as a combination of numbers and strings.

We usually divide our alphabet into classes:

- | | |
|----------------|------------------|
| 1. Letters | A-Za-z |
| 2. Operators | +-*/?.#%&! : ... |
| 3. Digits | 0-9 |
| 4. Quote marks | ' " ` |
| 5. Space | |

Alphabets: ASCII, EBCDIC, APL, Unicode, ...

If the character set is small enough (e.g., ASCII), we can determine the class of a character by indexing a table:

```
Char Class[128] =  
    { 0, ..., 1, ..., 5, ... };
```

```
Class['A'] = 1;
```

```
Class['0'] = 3;
```

```
Class['\t'] = 5;
```

Then decide what to do for the next character by its class. Languages with associative memory constructs (e.g., Perl, Snobol, or ICON) can do this using built in operators trivially.

Of course, use a symbolic value for 1, 2, ..., 5.

Standard coding style:

```
c = getchar();  
t = Class[c];  
switch (t) {  
    1. Letter: Starting a name; break  
    2. Operator: This character is the operator;  
       break  
    3. Digit: Starting a number; break  
    4. Quote: Starting a string; break  
    5. Space: Throw it away; break  
}
```

What really happens here?

Case 2 (A single character operator)

- a. Produce a token value and “type operator” as output
- b. Start over at beginning of program with the next character

Case 1 (Word or variable name)

We really want to keep reading characters of a name, form a complete name, and make certain that it is in a dictionary (i.e., symbol table). Now the character types take on a different significance.

```
L:  c = getchar();
    t = Class[c];
    switch (t) {
        cases 1 and 3: (Letter or Digit)
            name = name || c;
            goto L;
        default:
            Look up name in symbol table;
            Produce a new token;
            Put c back into input queue;
    }
```

Case 3 (Number, e.g., an integer)

Read successive digits and form number.

Enter number in a table of numbers (which may be the symbol table).

```
n = 0;
```

```
While ( t == 3 ) {  
    n = n*10 + ( c - '0' );  
    c = getchar();  
    t = Class[c];  
}
```

Output token value and class of number;

Put last c back into input queue;

In many languages, constructing a number can be done using a built in function or library function.

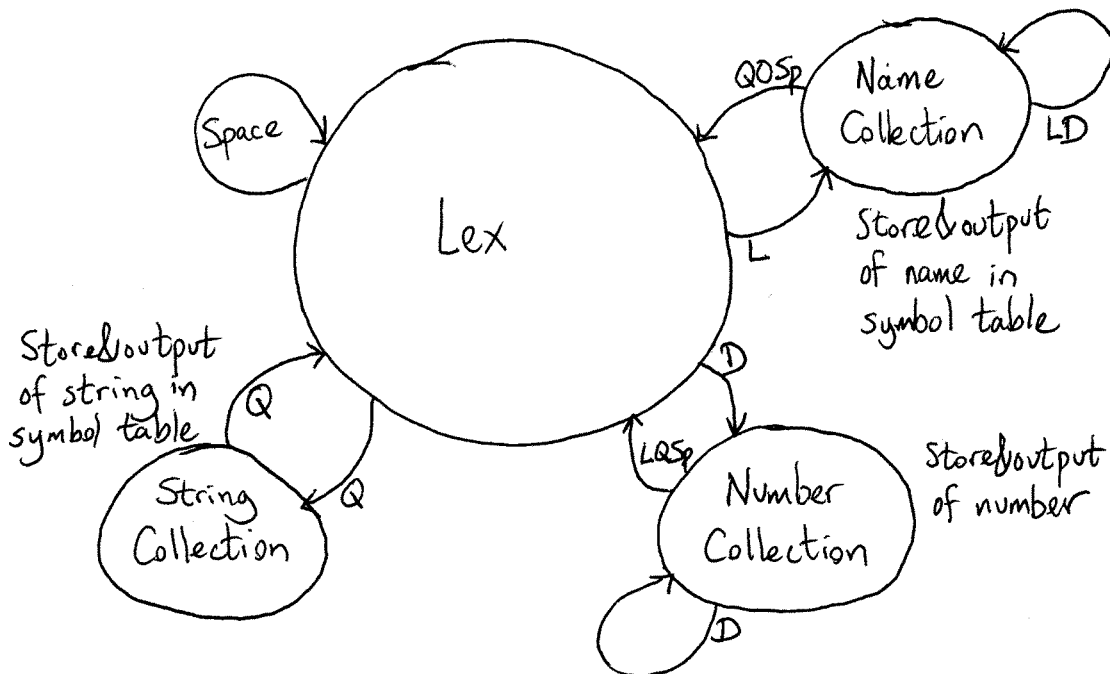
Throughout all of these programs we do the following:

1. Read a character and get its class
2. Branch to one of several small programs or code fragments
3. Loop back to one of several levels of read or branch calculation

We need to recognize that our process is in one of a number of states:

1. Initial: start of the program – ready for a new token
2. Identify an operator
3. Scanning a name: after first letter of name until its end
4. Scanning a number
5. Scanning a string

State transition diagram



This is easily stored and yields a simple program. But why not use a lexical analyzer generator instead... like *lex* or one of its cousins?

Note that *lex* has traditionally been C++ unfriendly whereas *flex* has had a C++ option for many years.

Pattern Matching

Unix users of `ed`, `vi`, `vim`, and similar editors might already know about one definition of regular expressions. See Chapter 6 of *Lex and Yacc*.

1. *Letters, digits, and some special characters* represent themselves
2. *Period* represents any character except a line feed
3. *Brackets []* enclose a character class. Anything in the class matches unless `[^]` is used: then anything outside of the class matches, e.g., `[^a]`. Hyphens inside of the `[]` allow for ranges, e.g., `[a-zA-Z]`.
4. Patterns ending in any of `*?+` match the pattern
 - * zero or as many times as possible
 - ? zero or one times
 - + one or as many times as possible
5. `^` before a pattern means match at the beginning of a line
6. `$` ending a pattern means match at the end of a line
7. Escape mechanisms:
 - a. `\` before a character, e.g., `\n` for linefeed or `\'`
 - b. “characters”
8. Multiple choice patterns: `(pattern|pattern|...)` matches one of the patterns.

Consider a C style comment:

```
/* single line comment */  
/* ... a multiline comment  
    ... */  
/**/          (a null comment)  
/*/ ... */    (odd beginning of a comment)
```

A single line comment might be described by

```
“/*”.*“*/”
```

The other three cases fail with this regular expression. All four can be described by the opaque regular expression

```
“/*”“/”*([^\*/][^*]“/”|“*”[^\/])*“*”“*/”
```

If you can explain this example in detail to someone else, you have mastered regular expressions far exceeding anything remotely reasonable.

How is an expression like this produced? Take one definition that is simple and make a regular expression. Then add a slightly harder case and modify the expression. Repeat until all of the cases are handled. Always do regression testing to make certain it is still correct for all cases, too.

Lex Programs

Input consists of up to three parts:

```
first part  
%%  
pattern      action  
...  
%%  
third part
```

The first part is optional and contains lines controlling certain internal to lex table sizes, definitions for text replacement, and global C code within `%{` and `%}` lines:

```
%{  
C code  
%}
```

The third part and its separator are optional, too. This is for C code that is taken as is.

The second part is quite line oriented. It starts at the first character and extends to the first non-escaped white space. Then an action appears after the white space. The longest expression that can be matched is used by lex. One line of C code can follow the

action, though multiple lines can be enclosed in brackets `{}`. There are no comments in Lex unless they are buried in the C code after the action.

Example: line numbering

```
%{
/* line numbering */
}%

%%

^.*\n    printf(“%d\t%s”, yylineno-1, yytext);
```

If this is stored in `exuc.l`, then it is compiled using

```
lex exuc.l
gcc lex.yy.c -ll -o exuc
```

The `-ll` is required and references the lex library. It has a default main program that just calls `yylex()`.

Lex has some global variables that are useful:

<code>yytext</code>	character vector with the match
<code>yylen</code>	integer length of <code>yytext</code>
<code>yylineno</code>	integer input line number
<code>yyval</code>	subvalue associated with <code>yytext</code>

Example: word count

```
%{
/* word count */

int  nchar, nword, nline;
%}

%%

\n      ++nchar, ++nline;
[^\t\n]+  ++nword, nchar += yyleng;
.        ++nchar;

%%

main() {
    yylex();
    printf(“%d\t%d\t%d\n”,
           nchar, nword, nline);
}
```

Grammar for Lexical Analysis

letter	[a-zA-Z_]
digit	[0-9]
letter_or_digit	[a-zA-Z_0-9]
white_space	[\t\n]
blank	[\t]
other	[^a-zA-Z_0-9 \t\n]

%%

```
==          return token(EQ);  
\<=       return token(LE);
```

```
{letter} {letter_or_digit}*    return name();
```

```
{digit}+          return number();
```

```
{white_space}+
```

```
{other}          return yytext[0];
```

%%

C functions for processing names and numbers.

This can be extended to cover a large subset of C and C++ fairly easily.

Screening for Keywords

Normally, the number of keywords in a language is relatively small, but even a 15-20 keywords makes for a large transition table.

```
#include "tokens.h"

char *keywords[] = {
    "int", "char", "double", ..., "" };
int tokens[] = { INT, CHAR, DOUBLE, ..., 0 };

int name( char *check ) {
    int i;
    for( i = 0; tokens[i]; i++ )
        if ( strcmp(check,keywords[i]) == 0 )
            return tokens[i];
    return IDENTIFIER;
}
```

If there are many keywords, a faster search algorithm is justifiable. A binary search or hashing method can be substituted. The key is to not store the special words in the symbol table unless a fast search method is used.

This trick also works with operators.

When Ambiguity is a ☺ in lex

Rules are usually highly ambiguous in lex, which resolves them using two rules:

1. lex always chooses the pattern representing the longest string match possible.
2. If multiple patterns represent the same string, the first one in the list is chosen.

Consider the two rules

```
int
[a-z]+
```

Then

```
integer  matches the second rule
int      matches the first rule
```

Always put specific rules first followed by general rules. Having too many specific rules will produce huge transition tables.

Conflicts are frequently encountered in lex. The order of the rules can take bizarre forms that are completely illogical at first glance (or even the seventeenth glance).

Lexing Complicated Numbers

Integers are easy, but adding the full definition of a floating point number (either real or complex) is much harder. Using the lex suggested definitions on page 25, we have

```
{digit}+
{digit}*“.”{digit}+d(“+”|“-”){digit}+
{digit}+“.”{digit}*d(“+”|“-”){digit}+
{digit}*“.”{digit}+d{digit}+
{digit}+“.”{digit}*d{digit}+
{digit}*“.”{digit}+
{digit}+“.”{digit}*
```

The actions are pretty simple: first translate the number into either an integer or a floating point number, e.g.,

```
atoi( yytext ) or atof( yytext );
```

Then enter the number in the symbol table.

Consider 1.23d-4 or .000123. Note which rule applies to each.

Floating Point and Integer lex Definitions

The previous page's numeric definition is awkward and goes against lex's philosophy of short and incomprehensible is better than long and readable.

```
-?(([0-9]+)|([0-9]*\.[0-9]+)([dD][+]?[0-9]+)?)
```

-?[0-9]+ is an integer with a possible unitary minus.

[0-9]*\.[0-9]+ is a real number with a required trailing digit after the decimal point. So 4.d-3 flunks even though it is a perfectly good floating point number.

The fix is

```
([0-9]*\.[0-9]+)|([0-9]+\.[0-9]*))
```

which leads to the one liner (albeit in a tiny font)

```
-?(([0-9]+)| (([0-9]*\.[0-9]+)|([0-9]+\.[0-9]*)))([dD][+]?[0-9]+)?)
```

You should verify this is a viable definition. It requires at least one digit before or after the digit. Try both definitions in a lex program.

Compare this to the definition on the previous page.

Format Systems for Describing Programming Languages

Context Free Grammars:

CFG Context Free Grammar

Letter \rightarrow A | B | C | D (rules 1.1-1.4)

Name \rightarrow Letter | Name Letter (rules 2.1-2.2)

BNF Backus-Naur Form

<Letter> ::= A | B | C | D

<Name> ::= <Letter> | <Name><Letter>

EBNF Extended BNF

[optional] and { repeat 0 or more times }

CFG starts at the top and tries to generate strings using the lower level definitions

BNF starts at the bottom of the rules and tries to make higher level symbols

EBNF can even describe itself, which BNF cannot do

grammar ::= rule { rule }

rule ::= NONTERMINAL ‘::=’ [formulation]
 { ‘|’ formulation }

formulation ::= symbol { symbol }

symbol ::= NONTERMINAL
 | TERMINAL
 | ‘{‘ formulation ‘}’
 | ‘[‘ formulation ‘]’

NONTERMINAL symbols are like variables in a programming language whereas TERMINAL symbols are like constants or key words. Space is usually ignored or used as a delimiter.

BNF for Numbers

Consider defining either integers, reals, or complex numbers. Only digits, +, -, d, (, and) are involved...

```
integer :    digit |
            integer digit
exponent :  d [+ -] integer | d integer
mantissa :  integer "." integer |
            integer "." |
            "." integer
real :      mantissa exponent |
            mantissa
complex :   "(" real "," real ")"
number :    integer | real | complex
```

Add a collection of actions for the parser. Note that it is not until the number is constructed that the actual type is known. Hence, a constant can start as an integer and get promoted to either real or complex. We actually have to keep the integer parts as strings until we can determine that they are not to the right of a decimal point (consider .0032 versus .32).

Note that numbers are usually constructed in the lexer, not the parser. However, there is nothing to stop the lexer from using a parser to construct numbers.

Ambiguities in Parsing ☹

Take BCD as input to the CFG example:

1. (init) Name \rightarrow
2. (2.2) Name Letter \rightarrow
3. (2.2) Name Letter Letter \rightarrow
4. (1.3) Name C Letter \rightarrow
5. (2.1) Letter C Letter \rightarrow
6. (1.2) B C Letter \rightarrow
7. (1.4) BCD

The other way around: Recognizer or parser approach

1. BCD (1.4) for D
2. BCD (1.2) for B
3. BCD (2.1) for B
4. BCD (1.3) for C
5. BCD (2.2) for BC
6. BCD (2.2) for BCD

Ambiguities should be avoided whenever possible ☺

Left and right recursion possible, too ☹²

Letter \rightarrow A | B | C | D

Name \rightarrow Letter | Name Name

Reverse Polish Notation (RPN)

See the following web sites for RPN summaries:

- http://glow.sourceforge.net/tutorial/lesson7/side_rpn.html
- <http://www.hpmuseum.org/rpn.htm>

This is an alternate notation that has been used in early pocket calculators (e.g., the HP 45 from 1972) that is an extremely efficient manner of representing expressions that need to be evaluated (or compiled).

Consider the example

$$2 + 3$$

which is represented in Polish notation as

$$2 3 +$$

The operators come after the operands. This is perfect for compiling.

Using standard notation, to evaluate

$$4 * (2 + 3)$$

you need to first calculate

$$2 + 3 = 5$$

and store the result in a temporary, then calculate

$$4 * 5 = 20$$

In Polish notation, the expression to be evaluated is

$$4\ 2\ 3\ +\ *$$

A stack is maintained in evaluating Polish notation and an operator results in immediate evaluation. The temporaries are always in the stack. You can evaluate expressions only as complicated as can fit into the stack.

So how do we parse Polish notation? Let's look at the notations for a simple example from the web page noted earlier:

standard: $[(9+3) * (4/2)] - [(3x) + (2-y)]$

Polish: $[(9\ 3\ +)\ (4\ 2\ /)\ *]\ [(3\ x\ *)\ (2\ y\ -)\ +]\ -$

Parsing, here are the steps. Try to draw the stack as it is generated and notice how a tree is naturally formed...

- Process 9 as a number, Push 9 onto stack.

- Process 3 as a number, Push 3 onto stack.
- Process + as an operator, Pop 3 as operand 2, Pop 9 as operand 1, Push + onto stack
- Process 4 as a number, Push 4 onto stack.
- Process 2 as a number, Push 2 onto stack.
- Process / as an operator, Pop 2 as operand 2, Pop 4 as operand 1, Push / onto stack.
- Process * as an operator, Pop / as operand 2, Pop + as operand 1, Push * onto stack.
- Process 3 as a number, Push 3 onto stack.
- Process x as a variable, Push x onto stack.
- Process * as an operator, Pop x as operand 2, Pop 3 as operand 1, Push * onto stack
- Process 2 as a number, Push 2 onto stack.
- Process y as a variable, Push y onto stack.
- Process - as an operator, Pop y as operand 2, Pop 2 as operand 1, Push - onto stack.
- Process + as an operator, Pop - as operand 2, Pop * as operand 1, Push + onto stack.
- Process - as an operator, Pop + as operand 2, Pop - as operand 1, Push - onto stack.

Production based syntax analysis

Number all of the productions. Use four variables to hold information about a parse tree:

- P Parse output tree
- S Parameter vector
- A Answer variable
- St Stack

P is represented as a Polish form $2 \times N$ matrix with the production number plus a character code.

a. For terminal symbols,

$$p[1,i] = \text{char. Code}$$

$$p[2,i] = 0$$

b. For nonterminal symbols,

$$p[1,i] = \text{production number}$$

$$p[2,i] = \text{length of production}$$

This example is a bit contrived.

Production	Rule	Action
1	Digit : "0"	0
2	Digit : "1"	1
	...	
10	Digit : "9"	9
11	Int : Digit	S[0]
12	Int : Int Digit	(S[1]*10) + S[0]

Example: The number 238 parses as follows:

<u>2</u> Digit 3	<u>3</u> Digit 4	<u>8</u> Digit 9
<hr style="width: 100%;"/>		
Int 11		
<hr style="width: 100%;"/>		
Int 12		
<hr style="width: 100%;"/>		
Int 12		

P = [51, 3, 11, 52, 4, 12, 57, 9, 12] 51 = "2", 52 = "3",
 [0, 1, 1, 0, 1, 2, 0, 1, 2] 57 = "8" (ASCII)

Program:

```
while ( # columns in P > 0) do
  if (P[1,2] == 0) then
    { St = P[1,1], St }
  else
    { Process production }
  P = P without first column
}
```

We process the productions using

```
S = first P[1,2] elements of St
St = all but what we put in S
switch( P[1,1] ) {
    case 1:  A = 0; break;
    case 2:  A = 1; break;
    ...
    case 10: A = 9; break;
    case 11: A = S[1]; break;
    case 12: A = (S[1]*10)+S[2]; break;
}
St = A, St
```

Note that in the end, we correctly store in our symbol table one entry for any integer 32:

0032, 032, and 32

However, in forming real or complex numbers, e.g.,

.0032, .032, and .32

we must produce three separate numbers as integers. Unfortunately, the integer string is needed until the actual type of number is determined and what part the integer is.

Mixing Definitions between yacc and lex

yacc produces a file `y.tab.h` with the token definitions in it. In the first part of a lex program add the line

```
#include "y.tab.h"
```

In `y.tab.h` is one line per token:

```
#define ICONSTANT 1  
#define DCONSTANT 2
```

You should always use the `%token` first for all of your tokens before using any of the following yacc statements:

```
%left  
%right  
%nonassoc
```

Then *you*, in effect, choose which tokens have symbolic values similar to each other.

yacc keeps generating token values every time it sees a new one, no matter how it is defined. You should preempt yacc's numbering scheme. It is also a quick way to check if you have unwanted, extra tokens.

One way to check is to predefine all of the tokens that you believe should exist for your language as part of your yacc input. Run yacc and see if it creates any new ones. If it does, determine if you forget some tokens or if your yacc input is in error. Iterate until yacc and you agree on the number of tokens. Predefine all of the tokens.

Alternately, predefine nothing and let yacc produce all of the tokens. Check them to see that all of the ones you expect are there. Then predefine them yourself as part of your yacc input.

The first method works better for me than the second one. I have to think about what I am doing in the first case, which is almost always a quicker way to get yacc's input right.

Two lex/yacc Examples

Roman numeral conversion

Roman numerals generally can be converted to decimal by adding up the values of the various characters present, using I=1, V=5, X=10, etc. Properly formed roman numeral strings always have larger valued characters on the left side of smaller values, with the exception of the digits 4 and 9, which, rather than require 4 repetitions of the 1 digit, are expressed with digit pairs in reverse order. Thus IX=9 and XL=40, etc.

Left recursion: long lex file, short yacc file

roman-left.l

```
I    { yylval= 1; return GLYPH; }
IV   { yylval= 4; return GLYPH; }
V    { yylval= 5; return GLYPH; }
IX   { yylval= 9; return GLYPH; }
X    { yylval= 10; return GLYPH; }
XL   { yylval= 40; return GLYPH; }
L    { yylval= 50; return GLYPH; }
XC   { yylval= 90; return GLYPH; }
C    { yylval=100; return GLYPH; }
```

roman-left.y

```
num : GLYPH    { $$=$1; }  
    | num GLYPH { $$=$1+$2; }  
    ;
```

Right recursion: short lex file, longer yacc file

roman-right.l

```
I  { yylval= 1; return GLYPH; }  
V  { yylval= 5; return GLYPH; }  
X  { yylval= 10; return GLYPH; }  
L  { yylval= 50; return GLYPH; }  
C  { yylval=100; return GLYPH; }
```

roman-right.y

```
{% int last=0; %}  
%%  
num : GLYPH    { $$=last=$1; }  
    | GLYPH num { if ($1>=last)  
                  $$=$2+(last=$1);  
                  else  
                  $$=$2-(last=$1); }  
    ;
```

The first example has the lexer catch roman numerals as special symbols, adding to the complexity of the lexer, but eliminating the need to check if a pair of characters is reversed when adding them up. This parser can scan the string from left to right and can use an efficient left-recursive grammar.

The second example scans the roman numeral string backwards, from right to left, remembering the last character it encountered, and conditionally subtracting rather than adding. The backwards scanning is performed in the yacc grammar by using a right-recursive form. This has the drawback of forcing yacc to "stack" up all the symbols before it can pop them off and process them. With a serious grammar, this stacking can get out of hand and cause the parser to fail, which is why left-recursion is usually a better idea with yacc.

How yacc works

Yacc transverses the rules in a highly parallel manner in order to produce a parser. The parser is a stack machine (actually a push-down automaton) consisting of

- a very large stack to hold the current states
- a transition matrix to derive a new state for every possible combination of the stack and the current input symbol
- user definable actions
- an interpreter to allow execution

The result is a function `yyparse()` that returns 0 or 1 if a sentence has been presented as input. `yylex()` is called repeatedly to get symbols.

This style of lexing and parsing is very efficient from the viewpoint of human time needed to generate a compiler. Remember, that one of the few things in life that cannot be recovered is your time.

It is not necessarily the most efficient from a CPU time viewpoint. However, it usually uses less CPU time than handwritten lexer-parser combinations.

Parse Functions

Input characters

↓

yylex() → next terminal symbol

↓

↓

state →

operation

...

(transition matrix)

(stack)

The

- current state (on top of the stack) and
- the next terminal symbol

select the next operation:

accept

This happens once when \$end is the next terminal symbol.

error

There is no transition in this state. This means the next terminal symbol should not be seen in the current state.

shift new-state

The terminal symbol is acceptable in the current state. The *new-state* is pushed onto the stack and becomes the current state. We have moved on in some configuration.

Reduce formulation-number

The state contains a complete configuration. As many symbols as the configuration has are popped off the stack. The uncovered state on top of the state becomes the new current state. The formulation is used before changing the stack and current state. The terminal symbol is re-used with the new current state.

goto state-number

This is the shift operation for the nonterminal generated in the *reduce formulation-number* operation.

Goto and shift are similar operations. Shift always uses and discards the next terminal symbol. Goto uses a nonterminal and leaves the terminal symbol on the stack.

`yyparse()` and `yylex()` need to use the same token definitions. This might seem difficult at first, but

```
yacc -d grammar.y
```

produces a file `y.tab.h` with the token definitions in it. In the first part of a lex program add the line

```
#include "y.tab.h"
```

There are some other useful options to `yacc`, e.g.,

- v produces a human readable set of tables
- t adds debugging directives for `gcc -g y.tab.c`

Generalized Error Recovery in yacc

We need a process that correctly identifies where an error occurs in the input. To be on the safe side, we assume that either input comes from either a file or a file passed through the C preprocessor, `cpp`.

Unfortunately, `cpp` is stored in random places on most Unix systems today. Even if you know where it is now, it will probably move real soon now. ☹

We define two routines to solve this problem:

- `yymark()`: This properly copes with output hints from `cpp` (file and line number). `yywhoseits` variables are modified inside of yacc in order to correctly identify the original file and line number for errors.
- `yywhere(char *s)` actually prints out the error message with a hint what the offending lexum is and where on the line. Professional compilers can identify which column the offender begins in. yacc is lucky to identify the correct line instead of missing by one (too many).
- No messages of the form, “Error in above program, rewrite program,” come out of `yywhere`.

Actual Code

See the *notes* web page for the code *yywhere.c*.

```
#include <stdio.h>

extern char    yytext[];        /* Problematic text */
extern int     yyleng;          /* Length of yytext */
extern int     yylineno;        /* Input line number */
extern int     yynerrs;         /* Total number of errors
*/

extern FILE*   yyerfp;          /* Error file descriptor */

static char*   filename = NULL; /* Input file name
*/

/*****
*****
*
* yymark() parses correctly input from the C preprocessor
(cpp)
*
*****
*****/

yymark() {

    if ( source != NULL ) free( source );
    source = (char *)malloc( yyleng+1, sizeof(char) );
    sscanf( " # %d %s", &yylineno, source );
}

/*****
*****
*
* yywhere() correctly prints out where we are for an
error,
* even if cpp is in use.
*
*****
*****/
```

```

yywhere() {
    int        colon = 0;        /* flag variable */
    char*      cp;
    int        len;

    if ( source && *source && strcmp(source, "\\\"")) {
        cp = source;
        len = strlen(cp);
        if ( *cp == '"' )
            cp++, len -= 2;
        if ( !strncmp(cp, "./", 2) )
            cp += 2, len -= 2;
        fprintf( yyerfp, "file %.*s", len, cp);
    }
    if ( yylineno > 0 ) {
        if ( colon ) fprintf( yyerfp, ", " );
    }
    if ( *yytext ) {
        for ( i = 0; i < 20; i++ )
            if ( !yytext[i] || yytext[i] ==
'\n' )
                break;
        if ( i ) {
            if ( colon ) {
                fprintf( yyerfp, " near
\"%.*s\", i, yytext );
                colon = 1;
            }
        }
        if ( colon ) fprintf( yyerfp, ": " );
    }
}

/*****
*****
*
* yyerror(s) tries to pinpoint where an error occurred.
*
*****
*****/

yyerror( char* s ) {
    fprintf( yyerfp, "[error %d] ", yyerrs+1 );
}

```

```
yywhere();  
fprintf( yyerfp, "%s\n", s );  
}
```

Errors should look something like

[error 1] src.c, line 1 near “/*”

[error 2] src.c, line 153001 near “bogussymbol”

One reason to print the error number out is to help with *cascading errors*. Another is to put a limit on the number of errors printed before giving up.

Desk calculating

Let us define the standard example combining lex and yacc: yadc... with doubles instead of ints.

```
/* Desc calculator - yacc */

%{
#include <stdio.h>
#define YYSTYPE double
%}

%token Constant
%left '+' '-'
%left '*' '/'
%%

line
: /* empty */
| line expression '\n'
  { printf("%g\n", $2); }

expression
: expression '+' expression
  { $$ = $1 + $3; }
| expression '-' expression
  { $$ = $1 - $3; }
| expression '*' expression
  { $$ = $1 * $3; }
| expression '/' expression
  { $$ = $1 / $3; }
| expression '=' expression
  { $$ = $1; }
| Constant
  /* $$ = $1 */
```

Note the definition of YYSTYPE. The default is an int, not a double. Almost anything can be defined if done correctly:

```
typedef char* CHAR_PTR
```

```
#define YYSTYPE CHAR_PTR
```

but not

```
#define YYSTYPE char*
```

```
%{ /* desk calculator - lex */

#include "y.tab.h"
#include <stdlib.h>
#include <math.h>

extern double yylval;
%}

digits    ([0-9]+)
pt        "."
sign      [+ -]?
exponent  ([eE]{sign}{digits})

%%

{digits}{pt}{digits}?{exponent}? |
{digits}?{pt}{digits}{exponent}? { yylval = atof(yytext);
                                   return Constant;
                                   }
[ \t]+   { }
\n       { return yytext[0]; }
.        { return yytext[0]; }
```

Both programs can be compiled and linked to get a working interactive desk calculator. *Warning:* Not all lexes will take this input without minor modifications, particularly the ()'s in the exponent and digits definitions.

OK... so how does this solve my immediate compiler problems? I typed

a - - b

and received an endless supply of worthless error messages back as a result. How do I get the compiler to do my error generation for me?

Add a rule with the reserved word **error** in it as close to a terminal symbol as possible and/or add **yyerror;** as a statement in an action. For example, add to the definition of expression the extra rule

```
expression : ...  
            | error
```

This stops some cascading error situations. Another method is

```
expression : ...  
            | Constant { yyerror; }  
            | error
```

Placement of the error symbols is guided by conflicting goals:

- as close as possible to the start symbol of the grammar,
- as close as possible to each terminal symbol,
- without introducing conflicts.

As you can imagine, doing all three at once is quite a feat. Many subtle errors in the grammar can be introduced using these three goals. But wait... there is more to this 😊

The following typical positions for error symbols is the result of the goals:

- into each recursive construct
- preferably not at the end of a formulation
- non-empty lists require two error variants, one for a problem at the beginning of the list and one for a problem at the current end of the list
- possibly empty lists require an error symbol in the empty branch

Consider the following table:

construct	EBNF	yacc input
optional sequence	$x: \{y\}$	$x: /* \text{null} */$ $ x y \{ \text{yyerrok}; \}$ $ x \text{error}$
sequence	$x: y \{ y \}$	$x: y$ $ x y \{ \text{yyerrok}; \}$ $ \text{error}$ $ x \text{error}$
list	$x: y \{ T y \}$	$x: y$ $ x T y \{ \text{yyerrok}; \}$ $ \text{error}$ $ x \text{error}$ $ x \text{error } y \{ \text{yyerrok}; \}$ $ x T \text{error}$

A yacc solution to the errors is the following:

```
%{
#define YYSTYPE double
void yyerror(const char *c);
%}

%token Constant
%left '='
%left '+' '-'
%left '*' '/'

%%

line:
    | line err '\n' { yyerrok; yyerror("invalid
expression"); }
    | line expression '\n' { printf("answer:%f\n\n",
$2); yylval = 0; }
    ;

expression: expression '+' expression    { $$ = $1 + $3; }
          | expression '-' expression    { $$ = $1 - $3; }
          | expression '*' expression    { $$ = $1 * $3; }
          | expression '/' expression    { if($3 == 0)
                                          yyerror("divide by 0
error");
                                          else $$ = $1 /
$3; }
          ;

          | expression '=' expression    { $$ = $1; }
          | Constant                    { /* $$ = $1 */ }
          | expression '+' err           { $$ = $1; }
          | err '+' expression           { $$ = $3; }
          | err '-' expression           { $$ = $3; }
          | expression '-' err           { $$ = $1; }
          | err '*' expression           { $$ = $3; }
          | expression '*' err           { $$ = $1; }
          | err '/' expression           { $$ = $3; }
          | expression '/' err           { $$ = $1; }
          | err '=' expression           { $$ = $3; }
          | expression '=' err           { $$ = $1; }
          ;

err:
    | error
    | err error
```

```
%%  
void yyerror(const char *c)  
{  
    printf("\'%s'\n",c);  
}
```

Besides the brute force method in expression, note the err definition, which works well and is concise.

Parsing Algorithms

Consider a simple grammar with terminal symbols

+*:=()

Basic grammar will be something along the lines of

Stmt : Stmt = Sum | Stmt ; Stmt

Sum : Sum + Term | Term

Term : Term * Factor | Factor

Factor : Name | Number | (Sum)

We will consider a collection of algorithms to produce a working parser, including

- Operator precedence
- Recursive descent
- Early's algorithm
- LL(k) parsers
- LR(k) parsers

There are variants, too, such as backtracking.

Directed acyclic graphs are important, too, in order to produce better code generators.

Operator precedence (the algorithm to beat)

1. Fast linear algorithm
2. Restricted grammars only
 - a. Productions must not have adjacent nonterminals (e.g.
Sum : - Sum Term
not legal if Term is a nonterminal, too)
 - b. Terminal symbols may occur only once in a grammar.

First, assign precedence to operators:

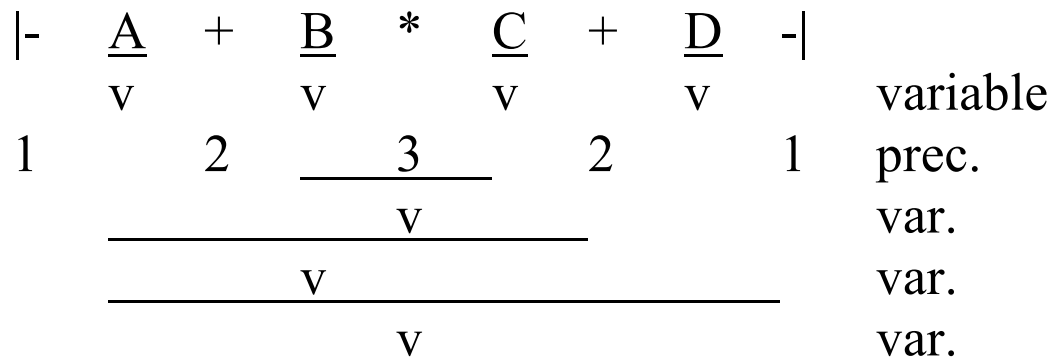
$$\begin{array}{ccccccc} A & + & B & * & C & + & D \\ & & 2 & & 3 & & 2 \end{array}$$

Higher numbered operators are done first. Look for peaks:

$$\begin{array}{l} O1 \vee O2 \vee O3 \quad (\vee = \text{variable}, O = \text{operator}) \\ P1 < P2 \geq P3 \quad (P = \text{precedence}) \end{array}$$

Then replace “ $\vee O2 \vee$ ” with “ \vee ” to get “ $O1 \vee O3$ ”.

The process is like drawing brackets all with the same name:



Paratheses do not fit into this pattern. Instead of precedence values for each operator, we use a relation:

- $<\bullet$ less than
- $O1 =\bullet O2$ equal
- $>\bullet$ greater than

We note that

- $+$ $<\bullet$ $($
- $($ $<\bullet$ $+$
- ... etc.

It is sometimes possible to find two functions f and g:

$$A \text{ prec } B \quad \text{if and only if} \quad f(A) \text{ prec } g(B).$$

The table form of our grammar is

	Stack	Input symbol							
	-	-	;	=	+	*	()	
0	-		Stop	<•	<•	<•	<•	<•	
1	-								
2	;		•>	•>	<•	<•	<•	<•	•>
3	=		•>	•>	<•	<•	<•	<•	•>
4	+		•>	•>		•>	<•	<•	•>
5	*		•>	•>		•>	•>	<•	•>
6	(<•	<•	<•	<•	<•	=•
7)		•>	•>		•>	•>		•>

The functional form can be

a	f(a)	g(a)
-	1	
-		1
;	4	3
=	4	5
+	7	6
*	9	8
(2	11
)	12	2

For reasonable grammars, both f and g can be found.

When using a table form, observe the following:

1. Only five possible values in each entry.
(Packing data is possible.)
2. Duplicate rows may exist.
(If many duplicates, store a pointer to the row.)
3. Break the grammar up into pieces where each piece has the function form.

Recursive Descent Parsing

One recursive function for each nonterminal, e.g.,

$$\langle \text{Name} \rangle ::= \langle \text{Letter} \rangle \quad (1)$$
$$| \langle \text{Letter} \rangle \langle \text{Name} \rangle \quad (2)$$

we have a function defined by

```
Proc Name() {  
    If ( Letter() ) then {  
        If ( Name() )  
            Then { Out(2) }  
            Else { Out(1) }  
        Return 1  
    }  
    return 0  
}
```

Function Name parses the longest name it can find starting at the current position in the input subject to

1. No name parsed.
 - a. Return 0
 - b. Input pointer and output are unchanged
2. Name parsed.
 - a. Return 1
 - b. Move input pointer to symbol after name
 - c. Add parse output

Issues that have to be tackled include

1. Syntax leads to Functions
2. Left Recursion
3. Determination
4. Efficiency

Turning BNF into programs is really easy (which is one reason why this is the method of choice in vendors' compiler writing groups).

$\langle W \rangle ::=$	$\langle X \rangle \langle Y \rangle \langle Z \rangle$	(1)
	$\langle X \rangle$	(2)
	$\langle X \rangle \langle Y \rangle \langle M \rangle$	(3)
	$\langle F \rangle$	(4)

```
Proc W() {
  If ( X() ) then {
    If ( Y() ) then {
      If ( Z() ) then { Out(1); Return 1 }
      If ( M() ) then { Out(3); Return 1 }
      --- Error, oops ---
    }
    Out(2); Return 1
  }
  If ( F() ) then { Out(4); Return 1 }
  Return 0
}
```

Left recursion is handled like

$$\langle \text{Name} \rangle ::= \langle \text{Letter} \rangle \mid \langle \text{Name} \rangle \langle \text{Letter} \rangle$$

1. Proc Name() {
 If (Name()) then ... oops, never returns ☹

2. Proc Name() {
 If (Letter()) then ... always returns ☺

There are multiple solutions, but two are

1. Use right recursion.

2. Find iterative way of doing it:

```
Proc Name() {  
    If ( Letter() ) then {  
        Out(1)  
        While ( Letter() ) { Out(2) }  
        Return 1  
    }  
    Return 0  
}
```

Early's Algorithm (CACM, February, 1970)

This algorithm is for small grammars, e.g.,

$$\langle E \rangle ::= \langle E \rangle + \langle T \rangle \quad (1)$$

$$\quad | \quad \langle T \rangle \quad (2)$$

$$\langle T \rangle ::= \langle T \rangle * \langle F \rangle \quad (3)$$

$$\quad | \quad \langle F \rangle \quad (4)$$

$$\langle F \rangle ::= a \quad (5)$$

(1) (2) (3)

Keep track of what to do with a “set of states” for each input symbol. Consider an example:

a	+	a
$\langle E \rangle ::= \cdot \langle E \rangle + \langle T \rangle$	$\langle T \rangle ::= \langle F \rangle \cdot$	$\langle E \rangle ::= \langle E \rangle + \cdot \langle T \rangle$
$\langle E \rangle ::= \cdot \langle T \rangle$	$\langle F \rangle ::= a \cdot$	
...
State set 1	2	3

The current symbol is associated with the *thing* after the bold dot.

4-tuples for each state:

1. P production number
2. j Position of dot
3. f Where in the input this first started
4. α List of symbols which can come after this production

Example: $\langle T \rangle ::= \langle T \rangle * . \langle F \rangle$
4-tuple: $\langle 3, 3, , \rangle$

Do one of three things for each state in the set for the current symbol:

1. Predictor Nonterminal $\langle T \rangle ::= \langle T \rangle * . \langle F \rangle$
We need to construct a $\langle F \rangle$
Add all productions about how to build a $\langle F \rangle$
2. Scanner Terminal $\langle T \rangle ::= \langle T \rangle . * \langle F \rangle$
We need a *
Check against input
3. Completer Nothing $\langle T \rangle ::= \langle T \rangle * \langle F \rangle$
We have made a $\langle T \rangle$ production !!!
Go back to where we put this production in and see what to do next

Space and Time Bounds:

For N = number of lexemes,

	Time	Space
In general	N^3	N^2
Unambiguous	N^2	N^2
“Bounded state”	N	N

For a general grammar,

The number of states for symbol I is $f(I)$ for each production or as many as $P \cdot f(I)$ states total (same production for each symbol).

We can do “search and add if not there” to a state set I in time P by organizing lists of states for each starting position and eliminating redundancies (so lists have at most P productions in them):

Predictor&Scanner	Do “s&a” only for each state (take $K \cdot (\#states)$)
Completer	Do “s&a” $\sim f(I)$ times (time $f(I)^2$)

Most of the time is spent in completer doing backtracking and copying state information (which is why this algorithm was superceded).

Unambiguous grammars:

Completer now takes time $f(I)$ because it can add a state to set I in only one way.

Example: (Palindromes) $\langle A \rangle ::= x \mid x \langle A \rangle x$
Takes N^2 time to produce $\sim N$ brackets
because $\sim(i/2)$ parses must be kept track
of for the i^{th} symbol.

Bounded Set Grammars:

No state bigger than some constant K .

Scanner and Predictor	time K
Completer	time K^2

$\Rightarrow \leq (K+K^2)N$ time total

LR(k) Parsers

- L Left to right input scanning
- R Rightmost derivation in reverse
- k Look ahead k items (if necessary)

Parser are generated through states for a grammar S.
Construct states using 3 operations.

1. Closure (I, a set of items)

If $A \rightarrow \alpha.Bb$ and $B \rightarrow \cdot\gamma \mid \cdot\delta$

then

$B \rightarrow \cdot\gamma$

$B \rightarrow \cdot\delta$

are both added to Closure(I), where

$\text{Closure}(I) = I \cup \{ \{B \rightarrow \cdot \dots\} \mid A \rightarrow \alpha.Bb \in I \}$.

2. Goto (I: set of items, X: symbol)

For all $A \rightarrow \delta.X\beta \in I$, define Goto as

Closure ($A \rightarrow \alpha X \cdot \beta$)

Intuitively, if I is the set of items valid for some prefix γ , then

$\text{Goto}(I, X) = \{ \text{items valid for prefix } \gamma X \}$.

Further, we can design DFA (discrete finite automata) for moving between states based on Goto).

3. Items (S' : augmented grammar)

Add new start symbol and production

$$S' \rightarrow S$$

where S is the old start symbol.

Set $C = \text{Closure}(S')$.

While sensible,

{ For all $I \in C$ and for any grammar symbol X ,
add $\text{Goto}(I, X)$ to C }

Items is the canonical collection of sets of LR(0) items for a grammar.

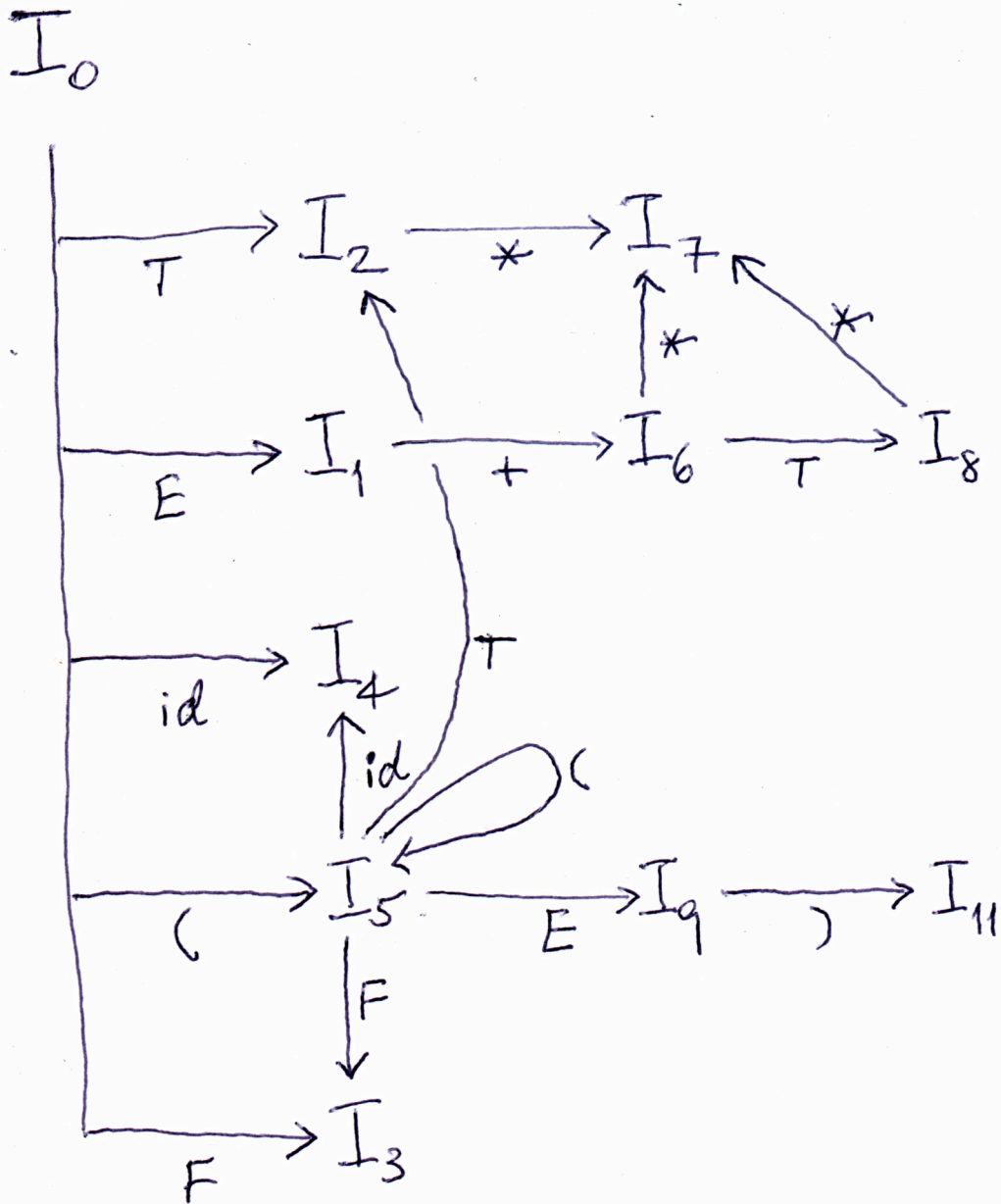
Example: $\langle E' \rangle ::= \langle E \rangle$
 $\langle E \rangle ::= \langle E \rangle + \langle T \rangle \mid \langle T \rangle$
 $\langle T \rangle ::= \langle T \rangle * \langle F \rangle \mid \langle F \rangle$
 $\langle F \rangle ::= \text{id} \mid (\langle E \rangle)$

The LR(0) parser has 11 states, which we can work out by hand fairly easily. However, computers are much better at creating the states.

Most parser generators look ahead 1 symbol and under special cases 2 symbols.

It helps to draw a diagram of the states and their relationships with each other.

Transition diagram



Items: $E' ::=$ $.E$
 $E.$
 $E ::=$ $.E+T$
 $E.+T$
 $E+.T$
 $E+T.$
 $.T$
 $T.$
 $T ::=$ $.T*F$
 $T.*F$
 $T*.F$
 $T*F.$
 $.F$
 $F.$
 $F ::=$ $.id$
 $id.$
 $.(E)$
 $(.E)$
 $(E.)$
 $(E).$

The states here are a bit mystifying to just look at (i.e., the rabbit pulled out of a hat syndrome).

$I_0: E' \rightarrow .E$
 $E \rightarrow .E+T$
 $E \rightarrow .T$
 $T \rightarrow .T*F$
 $T \rightarrow .F$
 $F \rightarrow .id$
 $F \rightarrow .(E)$

$I_1: E' \rightarrow E. \quad \text{Goto}(I_0, E)$
 $E \rightarrow E.+T$

$I_2: E \rightarrow T. \quad \text{Goto}(I_0, T), \text{Goto}(I_5, T)$
 $T \rightarrow T.*F$

$I_3: T \rightarrow F. \quad \text{Goto}(I_0, F), \text{Goto}(I_5, F)$

$I_4: F \rightarrow id. \quad \text{Goto}(I_0, id), \text{Goto}(I_5, id),$
 $\text{Goto}(I_6, id)$

$I_5: F \rightarrow (.E) \quad \text{Goto}(I_0,()), \text{Goto}(I_5,()),$
 $E \rightarrow .E+T \quad \text{Goto}(I_6,())$
 $E \rightarrow .T$
 $T \rightarrow .T*F$
 $T \rightarrow .F$
 $F \rightarrow .id$
 $F \rightarrow .(E)$

$I_6: E \rightarrow E+.T \quad \text{Goto}(I_1,+)$
 $T \rightarrow .T*F$
 $T \rightarrow .F$
 $F \rightarrow .id$
 $F \rightarrow .(E)$

$I_7: E \rightarrow T*.F \quad \text{Goto}(I_2,*), \text{Goto}(I_6,*),$
 $F \rightarrow .id \quad \text{Goto}(I_8,*)$
 $F \rightarrow .(E)$

$I_8: E \rightarrow E+T. \quad \text{Goto}(I_6,T)$

$I_9: F \rightarrow (E.) \quad \text{Goto}(I_9,E)$
 $E \rightarrow E.+T$

$I_{10}: T \rightarrow T*F. \quad \text{Goto}(I_7,F)$

$I_{11}: F \rightarrow (E). \quad \text{Goto}(I_9,))$

Creating LR(k) *by hand* for complicated grammars is not a good use of your time. Use a parser generator.

There are a number of variations of similar techniques, e.g., SLR(k), LL(k), etc.

Symbol Tables

This is a data structure that holds information about symbols discovered by the compiler, e.g.,

Names

- Value

- Type

- Size and shape

- Initial value (if any)

- Scope (local, global, mixed)

Constants:

- Numbers (integers, floating point, etc.)

- Strings

Classes

- All sorts of goodies

Functions

- Value

- Scope (local, global, mixed)

- Hints as to what it is and form

Externals

- Name

- Hints as to what it is and form

... (the list is almost endless)

The symbol table is usually a complicated data structure that is organized so that both the parser and code generator can easily access the information. The scoping information is particularly important to code optimization.

The design of a symbol table is usually done with personal taste involved. The implementation method used is frequently tied to the expected complexity of programs that will use it. Both have pluses and minuses.

Hashing, trees, and linked lists are common. Combinations and spaghetti coding structures are really common in commercial compilers, which is partly due to employee turnover and the long time between compiler rewrites from scratch.

You are free to be creative, but plan for flexibility and expandability as you write first a lexer, then a parser, then a code generator, then an optimizer.

There is a lot of information that is useful to store in a symbol table. However, before deciding what to store, it is useful to decide in advance whether to have one symbol table or a set of easily searched symbol tables.

The motivation for multiple symbol tables is twofold:

1. Use a small number of symbol types in a hashing function to get to like symbols.
2. For symbols defined multiple times in a file (thanks to procedures, classes, or inside of nested blocks), a tree of symbol tables makes it easy to find the right instance of a symbol.

For a tree form symbol table, there is a highest level symbol table. It has pointers to symbols and pointers to other symbol tables that are below it in the hierarchy. In lower symbol tables, pointers to even lower in the hierarchy symbol tables can exist. Each symbol table must have a pointer to its parent symbol table (or a NULL in the highest level one).

What sorts of information is potentially useful to sort the entries in a symbol table by?

- Symbols and identifiers
- Integers
- Floating point numbers
- Strings or characters
- [Constants]
- [Temporaries]
- Functions and procedures
- Classes

Inside the symbol table is a symbol entry containing all inclusive information about a single object:

- An integer (*whatami*) describing what this is.
- The actual object, which can be (possibly a union structure) one of the following:
 - A pointer to a character string containing a
 - Character (string) constant
 - Identifier, procedure, or function name
 - Printf string
 - An integer.
 - A double/float.
 - A pointer to a class object.
- Whether or not this is a constant.
- Something that indicates the valid scope of this symbol.

- A reference count (if zero after compiling, then it can be eliminated, preferably with a warning).
- What is the ideal memory type (e.g., register, stack, main memory).
- Memory address.
- For arrays,
 - The number of dimensions.
 - Lower and upper bounds on each dimension.
Note there can be a mix of the two types below:
 - Constants
 - Variables (which means that the object must be dynamically allocated).

The list goes on and on. ☹️

Classes and C style structs do not even fall into this system and usually require a linked list of symbol table entries or a dynamically allocated structure. At least this is simple to think about conceptually. 😊

Then there are lots of special cases... ☹️

Symbol Table Strategies I

Set up a hash table. The keys are the symbol names, stored as character strings. Each key is stored uniquely no matter how many times it is defined or used in the program(s) being compiled.

For each key, store

1. The symbol name
2. A pointer to a linked list of symbol table entries, one for each definition in the program(s) being compiled. Depending on the language, this might be at the procedural, class, or block level.

During the lexical analysis stage, just the keys will be created. You do not know enough to add the links. It is a bad idea to mix lexical analysis and parsing into the lexical analyzer. Keep the lexer simple and let the parser add complexity to your symbol table and fill in the links. Your parser will know if a key is

1. Already known in the current scope (and has a link already).
2. A new scope and definition is occurring (create a new link).
3. A redeclaration (an error).

During the parsing stage, you will need to fill in the scope in each of the links. This is tricky and error prone.

Symbol Table Strategies II

Keep a hierarchy of symbol tables stored as a tree with pointers up and down the tree. This requires that your lexer and parser cooperate on the symbol table creation. It is more complicated to build, but far easier to access and determine properties about any given symbol.

You will start with a symbol table for the global symbols. For each function or procedure, start a new symbol table as a leaf in the parent symbol table.

When searching for a symbol, start with the current symbol table, then search in its parent, ..., until the global symbols only tree is searched. Do not go down into another symbol table, only go up the tree.

Testing

You will want to test your implementation on really nasty examples, e.g., start with something equivalent to the following:

```
for ( int i = 0; i < 10; i++ ) {  
    int j = i;  
    for ( int i = 123; i < 200; i++ )  
        printf("(i,j) = (%d,%d)\n", i, j);  
}
```

and then make it much, much more complicated by adding

- a class
- a procedure
- more nonobvious code constructions
- an obvious error
- a subtle error.

Simple Code Generation

First, please review the class notes about the target machine way back at the beginning (pages 9-12).

Then look very carefully at the virtual computer that is defined in the f06.c file.

Memory is complicated. It can be referenced as

- Registers:
 - integer - R[constant]
 - double - F[constant]
- Main memory:
 - integer - Mem
 - double - FMem
 - string - SMem
- Stack memory starts at the end of memory and grows downward.
- Dynamic memory starts at the beginning of memory and grows upward.
- Static memory does not exist!

Many operations only work with registers, not with main memory. Hence, you have to move data to the right type of registers, do the operation, and then possibly store the result back into main memory. This is a nuisance, but is how many computer processors work. It also takes time. It is important to try to

minimize running time, so superfluous data movements should be minimized when practical, but is part of Optimized Code Generation, which comes later in the notes.

Dynamic Memory Allocation and Freeing

Dynamic memory is allocated from the beginning of main memory. Unlike a real computer system, do not worry about how freeing dynamic memory works or garbage collection. When memory runs out... oh, well.

Look in f06.c for routines to allocate and free memory. Do not write your own.

Timing Considerations

You will actually put two C statements per line. The first will increase a global variable F06_Time by given formulae based on the operation and memory access. So a typical line will look like

```
F06_Time += integer constant; generated code;
```

Yes, this is really ugly, but assembly language for a real computer is, too.

Note: OK, that is no excuse for perpetrating ugliness on the world, but assembly language is how most computers operate. Back in the middle 1960's, the Burroughs B5500 computers used simple Algol 60 as assembly language. Then they went out of business.
☹

The access rules for memory are the following:

Access type	Access time
R	1
F	2
Mem	20
Int /	19
Int %	20
Double /	38
Double %	39

Things accumulate. For example,

```
F06_Time += (20+1+1); Mem[R[2]] = R[1];  
F06_Time += (2+2+2); F[2] = F[1] * F[3];
```

It is OK to generate the timing in just this inefficient way. The F06_exit program in the f06.c file will print out the number of F06_Time ticks your code used and whatever return code your program returns.

Operations

In the following, items like R[1] can be replaced by any R[integer constant]. Similarly for F[2]. The Stack Register can be used as the index in Mem, but you cannot change the value of SR while doing so.

In general, you can work with one or two registers at a time or one register and memory. More complicated things are strictly forbidden. When in doubt, ask.

You will notice a lack of character strings and classes in the following examples. They are special and will be described separately. You will have the honor of writing some simple code to copy complex objects.

Manipulate the Stack Register SR:

```
++SR;  
--SR;  
SR += integer constant;  
SR -= integer constant;  
SR += R[1];  
SR -= R[1];
```

Manipulating the Frame Register FR:

Same as SR

FR = SR;

In the notes below, you may substitute FR for SR everywhere. You might want to use it to track where in the stack the floating point variables are in FMem.

Load (a register):

R[2] = R[1];

R[2] = Mem[loc];

R[2] = Mem[R[3]];

R[2] = Mem[R[3] \pm loc];

R[2] = Mem[SR];

R[2] = Mem[SR \pm loc];

F[1] = (double)Mem[loc];

F[1] = (double)Mem[R[3] \pm loc];

F[1] = (double)Mem[SR];

F[1] = (double)Mem[SR \pm loc];

F[3] = FMem[loc];

F[3] = FMem[R[3]];

F[3] = FMem[R[3] \pm loc];

F[3] = FMem[SR];

F[3] = FMem[SR \pm loc];

Store (a register):

Mem[loc] = R[2];
Mem[R[3]] = R[2];
Mem[R[3]±loc] = R[2];
Mem[SR] = R[2];
Mem[SR±loc] = R[2];
(double)Mem[loc] = F[1];
(double)Mem[R[3]] = F[1];
(double)Mem[R[3]±loc] = F[1];
(double)Mem[SR] = F[1];
(double)Mem[SR±loc] = F[1];
FMem[loc] = F[2];
FMem[R[3]] = F[2];
FMem[R[3]±loc] = F[2];

Data conversion (double to/from integer):

R[1] = (int)F[2];
F[2] = (double)R[1];

Strictly no Mem's.

Arithmetic (+-*/%):

R[2] = R[2] + R[1];
R[2] = R[1] + R[3];
R[2] = R[1] % R[3];
F[2] = F[2] * F[1];

Strictly no Mem's. What about $F[2] \% F[1]$?
This is optional, but easy to implement.

Comparison:

$R[0] = R[1] == R[2];$

$R[0] = F[1] < F[3];$

$R[0] = R[1] == 0;$

$R[0] = R[1] != 0.0;$

$==, !=, <=, <, >=, >, \&, |, !$ are legitimate comparison operators.

Strictly no Mem's anywhere and no F's on the left hand side of the equal sign. 0 and 0.0 are the only constants allowed.

Labels:

L123:

L followed by an integer.

Goto's:

goto L123;

Conditional Goto's:

if (R[1]) then goto L123;

No else clause and no comparisons.

Return:

Mess with the stack;
Free local variables;
Put the correct return value on stack if you are using that a stack method for returns.
Issue a *C return* statement

After the return, the stack may have to be manipulated in order to get the return value.

Early Exit:

call `F06_Exit(integer);`

Procedure/Function call:

Push arguments onto the stack;
Be defensive and push the number of arguments onto the stack so that it is the first thing popped off the stack.
Call internal name with no arguments using a *C call* statement.

See return for comments on return values.

Print/Read:

Use the built in functions in the `f06.c` file.

Constants:

Use your creativity to put constants in memory somewhere. Do not store the same constant twice.

Push and pop with respect to the stack:

```
Mem[--SR] = Mem[loc];  
Mem[loc] = Mem[SR++];
```

You must be careful of data types (integer and double). Do not put either strings or arrays on the stack without a lot of planning first.

Increment, decrement, and clearing

```
R[2]++ or ++ R[2]  
--Mem[loc];  
Mem[R[2]]++;  
F[1] = 0.0;  
R[2] = 0;  
Mem[loc] = 0;  
Mem[R[2]] = 0;  
Mem[R[2]±loc] = 0;  
Mem[SR] = 0;  
Mem[SR]±loc = 0;  
(double)Mem[loc] = 0.0;  
(double)Mem[R[2]] = 0.0;  
(double)Mem[R[2]±loc] = 0.0;
```

(double)Mem[SR] = 0.0;
(double)Mem[SR±loc] = 0.0;
FMem[loc] = 0.0;
FMem[R[2]] = 0.0;
FMem[R[2]±loc] = 0.0;
FMem[SR] = 0.0;
FMem[SR±loc] = 0.0;

No increment/decrement of doubles, but
clearing OK.

Shift left or right (R registers only):

R[2] << integer constant; (left)
R[2] >> integer constant; (right)

Class Objects and Character Strings

There are two major operations with strings:

String copy:

You need to write a built in procedure to copy one string to another location.

String comparison:

You need another built in procedure to do comparison of two strings (`==` and `!=` are sufficient).

The data must pass through the registers (R, F, or both) through loads and stores. You should write the simple C code yourselves using the rules given for code generation. You have to be careful how to deal with bytes after the end of string. One way is to allocate strings in units of 4 or 8 bytes.

A better approach is to work with blocks of memory (cf. `memcpy` and `memcmp` in C).

Sample Code

```
function integer main() {  
  
    integer a, b, c;  
    double x, y;  
  
    b := 2;  
    c := 3;  
    a := b + c;  
    y := 3.1415;  
    x := y * c;  
    return 0;  
}
```

We give two generated yourmain.h solutions. In the second one, printing of results is also included.

We allocate the variables on the stack:

a	Mem[SR]
b	Mem[SR+1]
c	Mem[SR+2]
x	Mem[SR+4] and Mem[SR+5]
y	Mem[SR+6] and Mem[SR+7]

Addressing x and y is a bit tricky since FMem[SR] is not the same location as Mem[SR] thanks to FMem

being associated with 64 bit objects while Mem is associated with 32 bit objects on many machines. (Of course, if all objects are 64 bits, then addressing is easy.)

FMem, SMem, and Mem all share the same space in memory. They are aliases for the same memory. However, indexing them by the same integer i leads to quite different memory locations due to the difference in size of the individual elements. On a typical PC or Mac, each element has size (in bytes) of

- 8 FMem
- 1 SMem
- 4 Mem

So, a conversion has to be made in order to access FMem or SMem correctly based on SR, which only works directly with Mem.

We can allocate doubles on the stack together, starting on an even Mem location (say, Mem[i]). Hence, we can set FR to be the index in FMem corresponding to Mem[i]. We can do something similar with strings, but we have to use one of the R registers as the string frame index.

There are two ways of generating code for this simple example:

1. Use just Mem, leading to really ugly and error prone simple C code.
2. Use Mem and FMem, leading to an extra register to use for the double variables. This also suggests that all doubles should be allocated together on the stack when a procedure is entered. *You should use the second method.*

Using the second approach, we allocate the variables on the stack as follows:

- a Mem[SR]
- b Mem[SR+1]
- c Mem[SR+2]
- x FMem[FR] (which is Mem[SR+4] and Mem[SR+5])
- y FMem[FR+1] (which is Mem[SR+6] and Mem[SR+7])

In both approaches, we have to first decrement SR by 8 integer words.

Generated Code Using Only Mem

```
int yourmain() {  
  
    /* Stack adjustment */  
    SR -= 8;  
  
    /* b = 2; */  
    R[1] = 2; F06_Time += (1);  
    Mem[SR+1] = R[1]; F06_Time += (20+1);  
  
    /* c = 3 */  
    R[2] = 3; F06_Time += (1);  
    Mem[SR+2] = R[2]; F06_Time += (20+1);  
  
    /* a = b + c; */  
    R[1] = Mem[SR+1]; F06_Time += (1+20);  
    R[2] = Mem[SR+2]; F06_Time += (1+20);  
    R[3] = R[1] + R[2]; F06_Time += (1+1+1+1);  
    Mem[SR] = R[3]; F06_Time += (20+1);  
  
    /* y = 3.1415; */  
    F[1] = 3.1415; F06_Time += (2);  
    *(double*)Mem[SR+6] = F[1]; F06_Time +=  
    (20+2);  
  
    /* x = y * c; */  
    R[1] = Mem[SR+2]; F06_Time += (1+20);
```

```

F[1] = (double)R[1]; F06_Time += (2+1);
F[2] = *(double*)Mem[SR+6]; F06_Time +=
(2+20);
F[3] = F[1] * F[2]; F06_Time += (2+2+2+4);
*(double*)Mem[SR+4] = F[3]; F06_Time +=
(20+2);

/* Stack adjustment */
SR += 8;

return 0;
}

```

Generated Code Using Mem and FMem + Printing Results

```

int yourmain() {

/* Stack adjustment */
SR -= 8;

/* Set Frame register to beginning of doubles */
FR = SR;
FR += 4;
FR = FR >> 1;

/* b = 2; */
R[1] = 2; F06_Time += (1);

```

```

Mem[SR+1] = R[1]; F06_Time += (20+1);

/* c = 3 */
R[2] = 3; F06_Time += (1);
Mem[SR+2] = R[2]; F06_Time += (20+1);

/* a = b + c; */
R[1] = Mem[SR+1]; F06_Time += (1+20);
R[2] = Mem[SR+2]; F06_Time += (1+20);
R[3] = R[1] + R[2]; F06_Time += (1+1+1+1);
Mem[SR] = R[3]; F06_Time += (20+1);

/* y = 3.1415; */
F[1] = 3.1415; F06_Time += (2);
FMem[FR+1] = F[1]; F06_Time += (20+2);

/* x = y * c; */
R[1] = Mem[SR+2]; F06_Time += (1+20);
F[1] = (double)R[1]; F06_Time += (2+1);
F[2] = FMem[FR+1]; F06_Time += (2+1+20);
F[3] = F[1] * F[2]; F06_Time += (2+2+2+4);
FMem[FR] = F[3]; F06_Time += (20+2);

/* Print results */
print_string( "a = " );
print_int( Mem[SR] );
print_string( "\nb = " );
print_int( Mem[SR+1] );

```

```

print_string( "\nc = " );
print_int( Mem[SR+2] );
print_string( "\nx = " );
print_double( FMem[FR] );
print_string( "\nx = " );
print_double( FMem[FR+1] );

/* Stack adjustment */
SR += 8;

return 0;
}

```

Notes:

- Recall that SR and FR are indices, not integer pointers.
- SR first has to be decremented by the full amount memory needed by the procedure's variables and temporaries.
- SR has to be restored to the correct value before the return, which may not be the same as the initial value of SR (this depends on how you implement return values).
- In the second case, FR is set to the beginning of all of the floating point numbers, which are grouped together on the stack. Accessing x and y is greatly simplified as a result.

Local Code Generation

Different types:

Arithmetic	$A[I] = B + C * D$
Control	if $X < Y$ then $P = Q$
Function calls	$F(X, Y+35)$
Declarations	double $A[236]$
Mixed arithmetic	double + int
Local optimization	$A = A + 1$

Arithmetic

$$A = (B+C) - (A*D)$$

Mov: $R[1] = A$
Mov: $R[2] = D$
Mul: $R[2] = R[2] * R[1]$
Mov: $R[0] = B$
Mov: $R[1] = C$
Add: $R[1] = R[1] + R[0]$
Sub : $R[1] = R[1] - R[2]$
Mov: $A = R[1]$

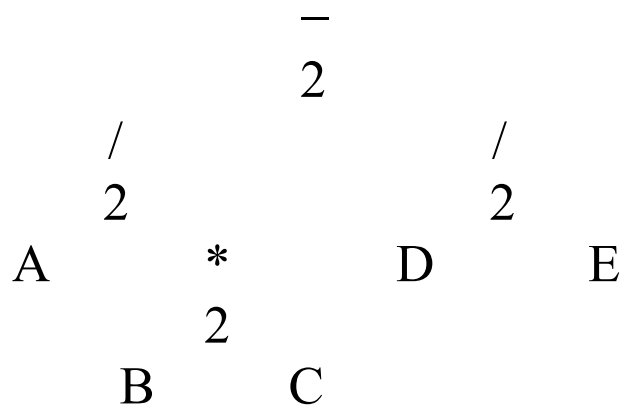
In reality, we have to replace A, B, C, and D with their locations in Mem, e.g., Mem[SR+3] for A, ..., Mem[SR] for D. The code is almost unreadable.

Two big questions:

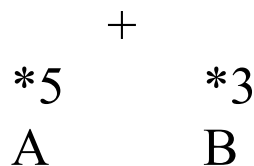
1. How many registers?
2. How do we assign them?

In a parse tree, assign the number of registers. Note: optimization techniques can change the number for a given leaf.

Consider $A/(B*C)-D/E$: in Polish notation, we have



Given a choice, *always* do the one with the most registers first:



A takes 5 registers

B takes $4=3+1$ registers

or

B takes 3 registers

A takes $6=5+1$ registers

What about leaves?

For machines which do $X = X \text{ op } Y$, one argument is also the result. However, consider $A/(B/C)$:

Mov: $R[1] = B$

Mov: $R[2] = C$

Div: $R[1] = R[1] / R[2]$

Mov: $R[2] = A$

Div: $R[2] = R[2] / R[1]$

Note: Result argument takes a register if

1. op is not commutative or
2. is a leaf (since it must hold result).

Assignment to Registers

Do a post order tree walk (i.e., parse the output order). For each internal node,

1. If the left argument is a leaf, issue code to move it to an available register (and mark the register as *used*).
2. Issue the following code:
 - a. Code for branch using most registers.
 - b. Code for branch using least registers.
 - c. Code for parent.
3. Mark register for the right argument *unused*.
4. Register for left argument is the result.

Example: AB-CD/*

	Stack	Code generated	Available registers
A	A V		012
B	A B V V		012
-	0 1 R R	R0←A R1←B R0←R0-R1	2
C	0 C R V		12
D	0 C D R V V		12
/	0 1 R R	R1←C R2←D R1←R1/R2	

*	0	$R0 \leftarrow R0 * R1$	12
	R		

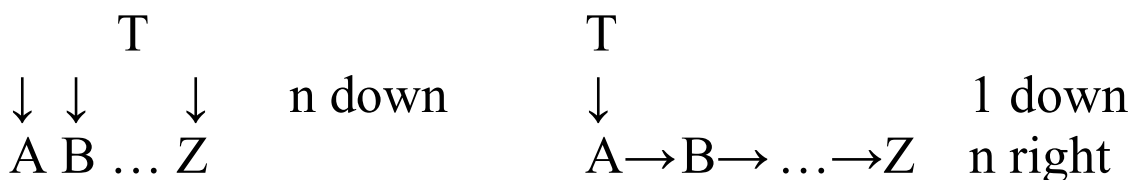
Further question: How to walk the tree:

1. Bottom up: Post-order
2. Top down: Pre-order

Walk(T) // Recursive function approach

1. If T is null, then return
2. Walk(Leftmost Child)
3. Walk(Next to Leftmost Child)
4. ...
5. Walk(Rightmost Child)
6. Return

Binary tree of a tree approach:



```

struct Tree { int  Type;
              int  Val;
              struct Tree  *Down;
              struct Tree  *Right; }

```

```

Walk(T)
  If T == 0 then return
  I = T→Down
  While(I≠0) {
    Process(I)
    Walk(I)
    I=I→Right
  }

```

Code Generation as a Tree Walk

We want a routine that generates code one node at a time:

```

Gen(struct Tree *T)
  A = T.Down      // 1st child
  B = T.Right     // 2nd child
  ...
  switch(T→Type) {
    case Var:     ...
    case Const:  ...
    case +:       ...
    case ITE:    ...
  }

```

Consider the runtime strategy of stack manipulation for the example $A+B*C$:

```
Mov: Mem[--SR] = A
Mov: Mem[--SR] = B
Mov: Mem[--SR] = C
Mov: R[1] = Mem[SR++]
Mul: R[1] = R[1] * Mem[SR++]
Mov: Mem[--SR] = R[1]
Add: Mem[SR] = Mem[SR] + Mem[SR++]
```

The switch cases of interest can be written like

```
Case Var: Write(" Mem[--SR] = ", T.Val)
```

```
Case *: Gen(A)
      Gen(B)
      Write("R[1] = Mem[SR++]")
      Write("R[2] = Mem[SR]")
      Write("R[1] = R[1] * R[2]")
      Write("Mem[--SR] = R[1]")
```

```
Case +: Gen(A)
      Gen(B)
      Write("R[1] = Mem[SR++]")
      Write("R[2] = Mem[SR]")
      Write("R[1] = R[1] * R[2]")
      Write("Mem[--SR] = R[1]")
```

This technique works well as long as cases are of the form

Case ...: Gen(A); Gen(B); ... Gen(...)
 Write(...)

This is post-order processing. We do not actually need to generate a formal parse tree. Sometimes we want to do something else, like

Case ...: Write(...)
 Gen(...)
 Write(...)
 Gen(...)

For example,

If $\underline{X < Y}$ then $\underline{P = Q}$ else $\underline{N = M}$
 A B C

Case ITE: I = Gen(A)
 T = NextLabel()
 Write(“ ”, I, “ ”, T)
 Gen(B)
 S = NextLabel()
 Write(“ Goto ”,S)

```
Write(T,“: ”)
Gen(C)
Write(S,“: “)
```

Another approach is to let Gen(...) return the generated code (or part of it). Then we would have instead

```
Case ITE:  CA = Gen(A)
           CB = Gen(B)
           CC = Gen(C)
           Result = Concatenation of CA, CB,
                   And CC
```

This requires extra bookkeeping. We need to do extra pointer chasing and we have to have the right type of concatenation function for linking code segments.

Many compilers skip the assembly code phase entirely and just generate machine code directly (or invisibly as load and go systems do). They use

```
struct INS { struct INS  *Next;
             int         Opcode;
             char        Lmode, Lreg, Lval;
             char        Rmode, Rreg, Rval; }
```

This is awkward to manipulate and usually results in a module that is hard to port to new machines.

What about register allocation? (top down approach)

1. Forget issue. Issue left branch first. Watch out for

$$A + B + C + D$$

which may use lots of registers.

2. Walk the tree first to collect useful information.

Arithmetic Expressions

Case +: $PA = \text{Gen}(A)$ The order is determined
 $PB = \text{Gen}(B)$ by # of registers used.
If (A not a register) { Put in $PA \rightarrow \text{Reg}$ }
If (B not a register) { Put in $PB \rightarrow \text{Reg}$ }
Write($PA \rightarrow \text{Reg}$, "=", $PA \rightarrow \text{Reg}$, "+",
 $PB \rightarrow \text{Reg}$)
Release register for B
Result = string designating value

Now let's consider some possibilities for the pointer to information about the value of B in the statement $PB = \text{Gen}(B)$:

<i>B</i>	<i>Type</i>	<i>Value</i>	<i>Comment</i>
A	Var	A	

35	Const	#35	
A[35]	Var	A(R3)	reg. containing subscript
+	Reg	R2	reg. containing X+Y
↓ ↓			
X Y			

We want a record for each value:

```
struct Val { char Type; // 'C', 'V', 'R', ...
             char Addr[?]; }
```

We need a stack of available registers and a routine to convert a value into a register:

```
ToReg( struct Val *P ) {
    If P→Type == 'R' then return
    R = top of stack, pop stack (check for error)
    Write( "R", R, "=", P→Addr )
    P→Type = 'R'
    P→Addr = 'R',R
    Return
}
```

Now there are two easy ways of doing $PB = \text{Gen}(B)$:

1. Storage in user stack space.
2. Storage in a heap.

In both cases you must explicitly free storage, which has to be done to avoid memory leak problems.

```
Gen( struct Tree *t, struct Val *P ) {
    A = 1st child
    B = 2nd child
    ...
    Case +: struct Val *PB
            Gen(P,A)
            Gen(PB,B)
            ToReg(P)
            Write(PA→Addr, "=",PA→Addr,
                  "+",PB→Addr)
            UnReg(B)

    Case [: // as in A[I]
            Gen(P,A)
            Gen(PB,B)
            ToReg(PB)
            P→Addr = P→Addr,"(",
                    PB→Addr,")"
```

Problem here... Register holding subscript should be released after usage.

How about A[I,J,K]? We might need to translate this into something wonderful like

$$A[(I*D2+J)*D3+K]$$

or something a lot uglier depending on how arrays are actually stored in memory (the calculation above assumes that A is stored as one large chunk of continuous memory).

Control Expressions

ITE situations need context in addition to what arithmetic expressions need. So, add another parameter to Gen:

Gen(Result, Node, Context)

The Gen code for A in

If { A } then { B } ...

is in a *branching* context. If A is false, then branch to some label X, otherwise to some label Z (the then clause). The expression in A can complicate branching, too.

Case And: In branching context, branch A
 False to X:

 Gen(\sim A, branch true to X)

 Gen(B, branch to X)

Case Or: Gen(?, A, branch true to Z)
 Gen(?, B, branch to X)

All these ideas apply to standard loop constructs as well (do, while, repeat, break, next, ...)... even to our class language.

Declarations: Runtime Environment

Example: register int **A()[100][10];

Questions: What is it?

How to address use of it?

In a language like C, what do we really know about a variable?

Storage class

register, auto, static, external

Type

int, long, float, double, char, struct, typedef,
union, ...

Access method

*A()[[]], ...

There are many hidden properties, some of which are machine dependent (standard today in *italics*):

Size

int 32 or 64 bits

long 32 or 64 bits

char 8 or 16 bits

float 32, 38, 64, or 80 bits

double 64, 80, or 128 bits

Data Access in C

Example: int *(*A[])(10)[3]; // general
 int *(*A[I](3,5)[J])[K]; // particular

To get this value, we have to

1. Get what A points at.
2. Add I to that and get what that points at
3. Call a function at that address with parameters 3 and 5.
4. Subscript result by J.
5. Get what that points at.
6. Subscript that by K.

(Hey, this is considered fun by some people.)

General
Subscr
 ↓ ↓
Indir 3
 ↓
Subscr
 ↓ ↓
Func 10
 ↓
Subscr
 ↓ ↓
Indir 1
 ↓
A

Particular
Subscr
 ↓ ↓
Indir K
 ↓
Subscr
 ↓ ↓
Func J
 ↓
Subscr
 ↓ ↓
Indir I
 ↓
A

Note: Value at every stage except the last is a pointer to an array or function. The arguments have been omitted.

In concept, edge vectors are a good method to implement something like

```
int *A[3][2];
```

So,

$$\begin{array}{c} A \\ \downarrow \\ *A \\ \downarrow \\ \text{edge vectors} \left\{ \begin{array}{l} A[0] \rightarrow (*A[0][0], *A[0][1]) \\ A[1] \rightarrow (*A[1][0], *A[1][1]) \\ A[2] \rightarrow (*A[2][0], *A[2][1]) \end{array} \right. \end{array}$$

To get $*A[I][J]$ we issue something like

```
R[1] = &Mem[A] // *A
R[2] = Mem[I]
R[1] = R[1] + R[2] // A[I]
R[1] = &Mem[R[I]] // *A[I]
R[2] = Mem[J]
R[1] = R[1] + R[2] // A[I][J]
R[1] = &Mem[R[1]] // *A[I][J]
```

Holy cow. However, edges require extra storage for the edge vectors.

Another possibility is to use storage by rows.

$$\begin{array}{l} A \\ \downarrow \\ *A \rightarrow \begin{array}{l} *A[0][0] \\ *A[0][1] \\ \dots \\ *A[2][1] \end{array} \end{array}$$

Now the code is

```
R[1] = &Mem[A] // *A
R[2] = Mem[I]
R[3] = n // n = #integers in last dim
R[2] = R[2] * R[3]
R[1] = R[1] + R[2] // *A[I]
R[2] = Mem[J]
R[1] = R[1] + R[2]
R[1] = &Mem[R[1]] // *A[I][J]
```

We trade space for the edge vector for extra code.
The exact amount is machine dependent.

Another example: $A[3][7][5]$

```
R[1] = (&A) + (7*5*n)*I // & A[I]
R[1] = R[1] + (5*n)*J   // &A[I][J]
R[1] = R[1] + n*K       // &A[I][J][K]
R[1] = &Mem[R[1]]       // A[I][J][K]
```

How much storage does a variable take?

```
char *A[3][4]
```

1 pointer sized unit: A is a pointer to a 3x4 array.

```
int A[3][4]
```

12 int sized units.

```
char A[3][4]([10])[20]
```

12 pointer sized units: A is a 3x4 array of function names, each function returns a pointer to a 10x20 array of chars.

Structures (and classes)

These can be addressed similarly to arrays.

	<u>Offset</u>	<u>Bytes</u>
Struct X { long A[10];	0	40
char B[3][5];	40	16*
struct X *Y };	56	4
		* padded

Total storage is 60 bytes on a 32 bit machine. It can be more on a 64 bit machine (machine dependent).

What should we store about variables?

How does

```
int A;          (*)
```

differ from

```
struct { int A; }
```

Can we regard (*) as a component inside an implicit struct?

1. Frame of a function
2. All of the static storage of a module

About a variable...

Name	text for name
Type	int/float/...
Strname	for a struct, its name
Size	number of bits per element
StorClass	reg/stack/static/heap
Scope	local/global
Home	what it is a component of
Offset	location within Home
Access	list of things required to get value

About a function...

Come in three forms: call by

1. value
2. address
3. name

Where do we put the parameters? The standard places are the stack and in the registers. Another approach is to put parameters first into a global array P:

P[0] = return address

P[1] = 1st parameter

P[2] = 2nd parameter

...

Examples: F(10,20) // 10,20 actual parms
and F(int A, int B) // formal parms
{ return A+B; }

Call by value: F(10,20)

```
P[1] = 10;   F:  P[1] = P[1] + P[2];  
P[2] = 20;   Goto *P[0];  
P[0] = &R;  
Goto F;  
R: ...
```

Call by address: F(X,Y+10)

```
P[0] = &R;   F:  P[1] = *P[1]+*P[2];  
P[1] = &X;   Goto *P[0];  
Tmp = Y+10;  
P[2] = &Tmp;  
Goto F;  
R:
```

Call by name: Parameters are evaluated only when used. In part, parameters are pointers to functions that return pointers to values. This is a very complicated method that has mostly died out. Do not use it.

Where do we put parameters?

1. In an array P
2. In registers (fastest)
3. On the stack (slowest)
4. At call (by address of call)

Example: F(A,B)

R[5] = &R

Call F

R: address of A

address of B

next instruction

Get address of A in F with R[1] = R[5]

Get value of A in F with R[1] = &Mem[R[5]]

5. Store parameters in function.

```
int F(A,B) // call by value example
```

```
F: return address
```

```
A: value of A
```

```
B: value of B
```

```
Code for F follows.
```

Call to F(X,Y) looks like

```
F_args[0] = &R;
```

```
F_args[1] = X;
```

```
F_args[2] = Y;
```

```
Call F;
```

R: ...

So,

```
F(A,3,5);  
F(B);      // actually does F(B,3,5)
```

Since 1st call sets parameters not passed in second call.

Commentary

Registers (standard place) – Fast, but cannot nest subroutine calls without saving registers (to stack).

At call, at function – Can nest, but not recursively.

On stack – Recursive, but slowest.

Mixed Arithmetic

Hierarchy	Type
1	char
2	int
3	float
4	double
5	complex

Do an operation on the highest mode, converting arguments when necessary (*coercion*). So,

```
int A; double B;  
A+B
```

means

1. Convert A to a double.
2. Do addition as a double.
3. Produce result as a double.

Coercion is always to a higher mode. No information is lost except when

1. int size is bigger than the mantissa size.
2. converting floating point number to int in a store operation (double \rightarrow float \rightarrow int).

Pointer Arithmetic

```
int *P, B[10];
```

P is a pointer to an int.

```
P = &B;
```

```
P++;    or    P = P + 1;
```

increments by the number of bytes per int.

```
Struct X { int A; char B[4]; } *P;
```

```
P++;
```

increments by the number of bytes per struct.

Optimization

Basic blocks

Consider a three block statement

$$X = Y + Z$$

which *defines* X and *uses* references Y and Z . The operator $+$ is generic. A name X is *live* (versus *dead*) if its value is used after that point (even in another basic block).

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without a halt or any chance of branching except at the end.

Given a set of 3-address statements, we determine basic blocks from *leader* statements, where a leader is

1. from the first statement.
2. any statement that is a target of any kind of branching statement.
3. any statement following any kind of branching statement.

For each leader, its basic block consists of the leader and all statements up to, but not including, the next leader or the end of the program.

Example: Compute string length.

```
n = 0;
For ( p = &s; *p++ != '\0'; ) n++;
```

1. n = 0	leader
2. p = &s	
3. t ₁ = *p	leader
4. p = p + 1	
5. if t ₁ == '\0' then goto 8	
6. n = n + 1	leader
7. goto 3	
8. ...	leader

Local transformations on basic blocks

1. Common subexpression elimination

a = b + c		a = b + c	b live, so
b = a - d	⇒	b = a - d	we cannot
c = b + c		c = b + c	eliminate
d = a - d		d = b	b+c

2. Dead code elimination

$$x = y + z$$

However, if x is dead, then we can eliminate entire statement without changing the value of the basic block.

3. Renaming temporary variables

Given

$$t_i = b + c$$

we can rename it to

$$t_j = b + c \quad (i \neq j)$$

and all occurrences of t_i to t_j after that in the block. A normal form basic block is one in which all statements defining temporaries define unique temporaries.

4. Interchanging statements

Suppose

$$t_1 = b + c$$

$$t_2 = x + y$$

Are adjacent and the names are disjoint. Then we can interchange the statements without affecting the results.

Note: Normal form basic blocks allow the maximal number of interchanges.

5. Algebraic transformations

Simplify expressions

$$x = x \pm 0 \quad \rightarrow \quad \text{no work}$$

$$x = x * 1 \quad \rightarrow \quad \text{no work}$$

$$x = y^2 \quad \rightarrow \quad x = y * y$$

Flow Graphs

Flow control can be added by constructing a directed graph. Each node is a basic block. The reason for an edge's existence can only be determined by looking at the jump statement at the end of a block.

A *loop* is a collection of nodes in the flow graph such that

1. There exists a path wholly within a collection of nodes.
2. There is no node which is the only way to get from any node outside the collection into the collection.

A loop containing no loops is an *inner loop*, else it is an *outer loop*.

Data Representation for Basic Blocks

There are many forms, e.g.,

1. Linked list of assembly statements or quads (frequently a doubly linked list).
2. Dynamically allocated structure with the count of the number of statements, a pointer to leader, and pointers to both the predecessor and the successor blocks.
3. DAGs
4. Some combination of the above.

Branches should refer to the name of a block, never to a leader. This way if the leader is eliminated or moved, we do not have to track down a collection of branch statements and modify them.

DAGs

Nodes should have the following labels:

1. Leaves are labeled by unique identifiers (either variable names or constants). Whether it is a value or a storage location is determined by the operator.
2. Interior nodes should use the operator as the label.

3. Optionally nodes are given a sequence of identifiers as labels. This is useful when computing values because a label is assumed to be that value.

Notes:

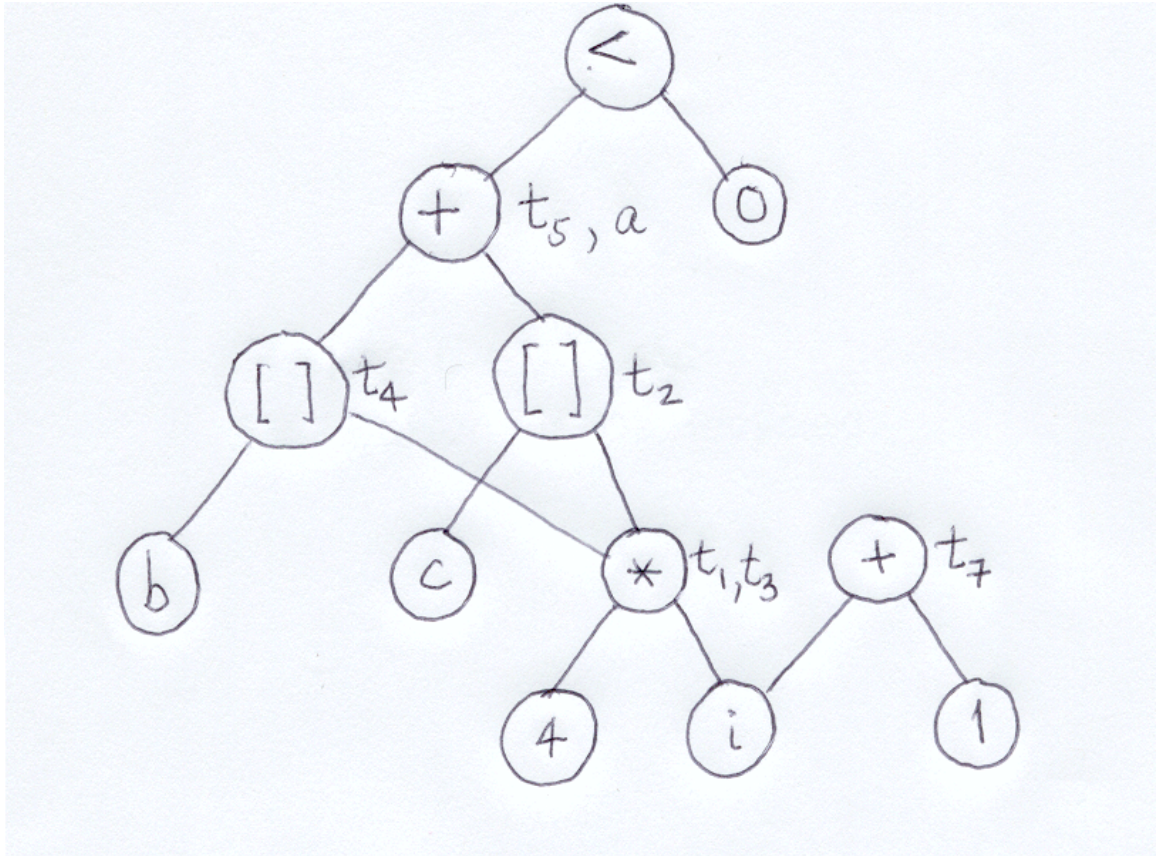
1. You have to have a DAG per basic block.
2. This is not related to the flow graph, but complementary.
3. If there is no common subexpression in a basic block, then we normally get a tree, not a DAG.

Example: $i = 1$; repeat $a = b[i] + c[i]$; $i++$;
 until ($a < 0$);

B1: $I = 1$
B2: $t_1 = 4 * i$
 $t_2 = b[t_1]$
 $t_3 = 4 * i$
 $t_4 = c[t_3]$
 $t_5 = t_2 + t_4$
 $a = t_5$
 $t_7 = I + 1$
 $i = t_7$
 if $a < 0$ goto B2

Inspecting the DAG is illuminating.

DAG for B2:



Transforming a Basic Block into a DAG

We want a label for each node, either an operator (from an interior node) or an identifier/constant (from a leaf). For each node, we want a list of attached identifiers (no constants allowed) and an empty list is OK.

Init: No nodes

Basic block statements are in the form

- i. $x = y \text{ op } z$ (this include if $y < z$
goto blah)
- ii. $x = \text{op } z$
- iii. $x = y$

Algorithm: For each statement in a basic block,

1. If $\text{node}(y)$ is empty, then create a leaf y and let $\text{node}(y)$ be this node. For case (i), repeat for $\text{node}(z)$.
2. Case (i): Determine if node labeled op exists with left child $\text{node}(y)$ and right child $\text{node}(z)$. If not, create such a node and call it n . Similarly for Cases (ii) and (iii).
3. Delete x from the list of attached identifiers for node x . Append x to the list of attached identifiers for node n in 2 and set $\text{node}(x)$ to n .

There are some assumptions in this algorithm, e.g.,

1. Appropriate data structures exist to create nodes with distinctive left/right children. Labels and the linked list of attached identifiers must be included in the structure.
2. All names and constants used in a DAG should be in a list.
3. There exists some $fn(identifier)$ that returns the most recently created node with label identifier.

There is some useful information that we get, e.g.,

1. Common subexpressions are automatically detected.
2. All identifiers/constants used in each block are detected.
3. Every statement that computes a value that can be used outside of the block is easily identified.
4. A simplified 3-address block can be reconstructed from a DAG with common subexpressions eliminated.

Array Pointers and Function Calls

When any of these can be on the left hand side, we need an *eliminated flag* for each node. This allows us to exclude certain nodes from common subexpression collapsing. Array indexing and pointer chasing can be loosely used for eliminating some node, but not all. A function call usually stops all collapsing in a block.

Dependent Order

When reassembling a basic block, we may have to require an ordering of statement execution. Certain edges in the DAG that were introduced must enforce the following rules when reordering the code:

1. Uses of an array may not cross each other.
2. No statement may cross a function call or pointer assignment.

Peephole Optimization

Look at a small number of statements (quad or 3-address), the *peephole*, and reduce the number of instructions or use faster ones. The peephole moves across the file providing a window. Multiple passes at the code is preferable.

Redundant code

Look for

```
Mov    B,A  
Mov    A,B
```

And eliminate the second statement. This only works when the second statement is unlabeled. This is also unnecessary if a DAG approach is used.

Unreachable code

Any basic block that does not have any directed edges to it in a flow graph can be eliminated. Alternately, any unlabeled statement after an unconditional branch can be eliminated.

Flow of control

Branches to unconditional branches can be converted into a single branch. When a label no longer has any branches to it, then the label can be removed.

Algebraic simplification and machine idioms

Purge null arithmetic statements. Take advantage of increment instructions and any other hardware specific instructions.

Reduction in strength

Replace expensive operations by cheaper ones, e.g.,

x^2	→ $x * x$
$\text{int} * 2^n$	→ shift left by n bits
$\text{float} / \text{constant}$	→ $\text{float} * (1/\text{constant})$
...	→ the list is almost endless

Appendix 1: Getting lex to Do *include* Files

First define a stack mechanism so that you can go back to the earlier file (or files).

```
%x INCLMODE

%{

#include <stdlib.h>

typedef YY_BUFFER_STATE ElementType;

struct Node
{
    ElementType element;
    struct Node * next;
};

struct Node *head = NULL;
```

Now add code to determine if the stack is empty.

```
int IsEmpty()
{
    if(head == NULL) return 1;
    else return (head->next == NULL);
}
```

You need to be able to push something onto the stack

```
void Push(ElementType element)
{
    struct Node *tmpNode;

    /* initialize the stack if it's empty */
    if(head == NULL){
        head = malloc(sizeof(struct Node));
        if(head == NULL){
            fprintf(stderr, "out of memory\n");
            exit(1);
        }
        head->next = NULL;
    }
    tmpNode = malloc(sizeof(struct Node));
    if(tmpNode == NULL){
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    else {
        tmpNode->element = element;
        tmpNode->next = head->next;
        head->next = tmpNode;
    }
}
```

You need to be able to pop something off of the stack

```
ElementType Pop()
{
    struct Node * firstCell;
    ElementType value;

    if(IsEmpty()){
        fprintf(stderr, "Empty stack\n");
        exit(1);
    }
    else {
        firstCell = head->next;
        head->next = head->next->next;
        value = firstCell->element;
        free(firstCell);
        return value;
    }
}

%}
```

You need lex rules process the file name especially since a file name does not correspond to the definition of an IDENTIFIER.

```
%%
...
^include                { printf("KEYWORD:\t\t%s\n",
                               yytext);BEGIN INCLMODE; }
...
<INCLMODE>[ \t]*       /* ignore the white space after the
include keyword */
<INCLMODE>[^ \t\n]+/;   { /* get the include file name */

    Push(YY_CURRENT_BUFFER);

    yyin = fopen( yytext, "r" );

    if ( !yyin )
        fprintf( stderr, "Cannot open file %s\n",
                yytext );

    yy_switch_to_buffer(
        yy_create_buffer( yyin, YY_BUF_SIZE ) );

    BEGIN(INITIAL);
}

<<EOF>> {
    if(IsEmpty())
    {
        yyterminate();
    }
    else
    {
        yy_delete_buffer( YY_CURRENT_BUFFER );
        yy_switch_to_buffer(Pop());
    }
}
```

You need a main program that always works with files and tells lex what to start with.

```
%%  
  
int main(int argc, char **argv)  
{  
    if(argc > 1) {  
        FILE * file;  
        file = fopen(argv[1], "r");  
        if(!file) {  
            fprintf(stderr,  
                "Could not open file %s\n",  
                argv[1]);  
            exit(1);  
        }  
        yyin = file;  
        yylex();  
    }  
    else {  
        printf("Usage is: %s <input file>\n",  
            *argv);  
    }  
    return 0;  
}
```

You can easily modify the main program to work with stdin instead of producing an error message. The include mechanism needs to keep track of file names and line numbers for error messages while parsing.

The definition of a file name is independent of anything in the language (especially the definition of an IDENTIFIER). Using include *whatever_including_white_space*; as a file name is general enough. Not allowing a semicolon in the file name should not be too much of a restriction.

Appendix 2: Making a C++ Parser Using lex and yacc

Flex has a C++ option, but lex does not. Yacc does not know how to deal with a C++ lex program unfortunately.

So, we generate a standard C program using lex and a parser in C++.

When you then link your application, you may run into some problems because the C++ code by default won't be able to find C functions, unless you've told it that those functions are extern "C".

To do so, make a C header in YACC like this:

```
extern "C"
{
    int yyparse(void);
    int yylex(void);
    int yywrap()
    {
        return 1;
    }
}
```

If you want to declare or change `yydebug`, you must now do it like this:

```
extern int yydebug;
```

```
main()
{
    yydebug=1;
    yyparse();
}
```

This is because C++'s One Definition Rule, which disallows multiple definitions of `yydebug`.

You may also find that you need to repeat the `#define` of `YYSTYPE` in your Lex file, because of C++'s stricter type checking.

To compile, do something like this:

```
lex bindconfig2.l
yacc --verbose --debug -d bindconfig2.y -o
    bindconfig2.cpp
gcc -c lex.yy.c -o lex.yy.o
g++ lex.yy.o bindconfig2.cpp -o bindconfig2
```

Due to the `-o` statement, `y.tab.h` is now called `bindconfig2.cc.h`, so take that into account.

Keeping lex and yacc Compatible

Checklist of Mandatory Things to Have:

1. Each lexer rule must exhibit a return value unless it is to be ignored. (can return literal or a token)
2. Implement yywrap function in lexer.
3. Must have union with data type variables in parser.
4. Must define a starting rule for parser to look for.
5. All tokens your lexer can return must be defined in the parser in terms of a union variable.
6. All parser productions that evaluate to a value must be typed.
7. Main function defined in parser must call yyparse. (preferably infinitely until EOF or exit)

8. Error handling after the `yyparse` call.
(`yyclearin`, `yyerror`, etc...)

9. Implement `yyerror` function in parser.

Tips for Happy Parsing:

Make your lexer as atomic as possible and return tokens whenever possible.

1. Do not match any "high level" rules in the lexer, leave those to the parser.
2. In the parser's error handling, try to give useful output to tell the user which input broke the parser rule.
3. Try to implement error handling such that the parser can continue after encountering errors.

A Makefile is especially useful, particularly when piecing codes together that use multiple stages in multiple files.