Compilers for Algorithmic Languages

University of Kentucky CS 441G
Fall, 2002

Craig C. Douglas

douglas@ccs.uky.edu
321A McVey Hall
+1-859-257-2326

Modules of a Compiler

1. Symbol table
2. Lexical analysis
3. Parsing
4. Optimize
    a. Local
    b. Global
5. Code generation
6. Assembly
7. Loader/linker
8. Errors

Symbol Table

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

1. Names
    a. Value
    b. Type
    c. Size and shape
    d. Initial value (if any)
    e. Scope (local, global, mixed)
2. Constants:
    a. Numbers
    b. Strings
3. Functions
    a. Value
    b. Scope (local, global, mixed)
    c. Hints as to what it is and form
4. Externals
    a. Name
    b. Hints as to what it is and form
5. …

The 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. Part has to do with turnover of employees.

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.

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*: +, -, if, //, :=, !, ;
3. *Numbers*:
    a. Integers: strings of digits
    b. Reals: Fraction + Exponent
    c. Complex: two reals
4. *Character vectors* or *strings*: surrounded by quotes
5. *Space*: usually used as a delimiter and thrown away.

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 new token, put c back on
            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 *= 10 + ( c – ‘0’ );
    c = getchar();
    t = Class[c];
}
```

Output token value and class of number.
Put last c back on input queue.

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

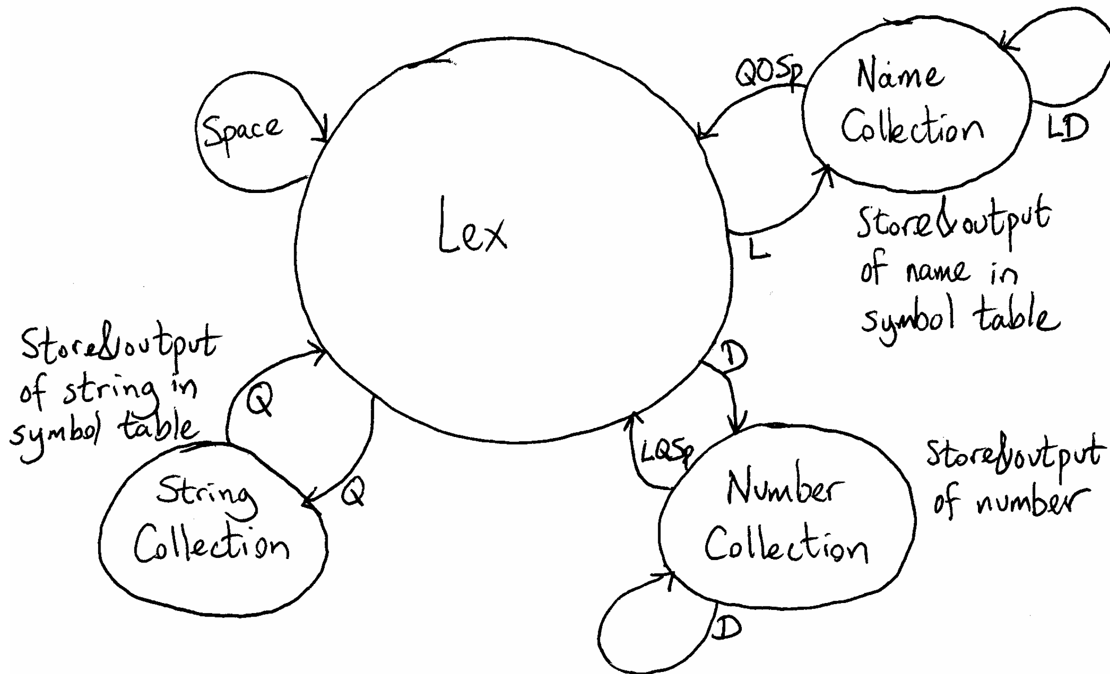
Throughout all of these programs we do the following:

1. Read a character and get its class
2. Branch to one of several small programs
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:

```
struct { int next, act, read; } tran[i][j];
    // Here i refers to the current state and j is
    // for the class of character
    // tran.next[i][j] = next state
    // tran.act[i][j] =  action to take
    // tran.read[i][j] = 1 if read new character

state = 1;
rd = 1;
while ( true ) {
    if ( rd ) { c = getchar(); t = Class[c]; }
```

```
        ac = tran.act[state][t];
        rd = tran.read[state][t];
        state = tran.next[state][t];
        switch ( ac ) {
            case 1:   name = c; break
            case 2:   name ||= c; break
            …
        }
}
```

<u>Different Methods of Compiling</u>

Multi-pass

1. Do lexical analysis: Text → *Lex* → Tokens
2. Do parsing: Tokens → *Parse* → Parse Tree
3. Do code generation: Parse Tree → *Gen* → Assemby or object code

Each pass is done sequentially. Data is typically stored on disk.

Co-routines

Spawn 3 processes with pipes in between

Input file → *Lex* → *Parse* → *Gen* → Output file

A pipe is really just a queue.

As soon as process 1 (Lex) has written some tokens, process 2 (Parse) can wake up and read them, …

Conventional organization

Parse calls Lex as a subroutine to get another token.

```
int state, place;
switch (place) {
	case 1: initialize; break;
	case 2: … ; break;
	case 3: … ; break;
	…
}
```

Three pitfalls to watch out for:

1. All variables must be static, not on the stack.
2. Switch beginning may involve jumping into implied loops from previous call
3. Some compilers will optimize temporaries into registers or onto the stack

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.

Ambiguities in Parsing ☹

Take BCD as input to the CFG example:

1. (init) Name →
2. (2.2) Name Letter →
3. (2.2) Name Letter Letter →
4. (1.3) Name C Letter →
5. (2.1) Letter C Letter →
6. (1.2) B C Letter →
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 → A | B| C | D
Name → Letter | Name Name

Pattern Matching

Unix users of ed, vi, vim, and similar editors might already know about one definition f 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. * or ? ending a pattern means that a match can occur zero or as many times as possible; + ending a pattern means 1 or more matches
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:

yytext	character vector with the match
yyleng	integer length of yytext
yylineno	integer input line number
yylval	subvalue associated with yytext

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 complicted numbers

Integers are easy, but adding the full definition of a floating point number (either real or complex) is much harder. Using p. 23's definitions, 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.

Actually defining numbers using a small parser makes a lot of sense. This is commonly done using BNF.

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.

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 2xN matrix with the production number plus a character code.

a. For terminal symbols,

  p[1,i] = char. Code

  p[2,i] = 0

b. For nonterminal symbols,

  p[1,i] = production number

  p[2,i] = 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:

```
  2           3           8
Digit 3     Digit 4     Digit 9

Int 11

        Int 12

            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”

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.

Checking a grammar

All nonterminals must satisfy

1. Each must have at least one rule.
2. Each must be reachable from the start symbol.

Consider a simple yacc file: grammar.y

```
%token  IDENTIFIER

%%

expression
    :   expression '+' IDENTIFIER
    |   IDENTIFIER
```

We test it with either

```
yacc grammar.y
yacc –v grammar.y
```

Nothing seems happen in either case => no errors in the grammar. In both cases, a file y.tab.c is created that holds the parser. In the second case, an extra file, y.output, is created that contains a lot of state information.

For our short grammar, y.tab.c contains

- o 344 lines of commented code
- o 4 terminals, 2 nonterminals
- o 3 grammar rules, 5 states

Holy cow!

y.output

```
   0  $accept : expression $end

   1  expression : expression '+' IDENTIFIER
   2             | IDENTIFIER

state 0
        $accept : . expression $end  (0)

        IDENTIFIER  shift 1
        .  error

        expression  goto 2


state 1
        expression : IDENTIFIER .  (2)

        .  reduce 2


state 2
        $accept : expression . $end  (0)
        expression : expression . '+' IDENTIFIER  (1)

        $end  accept
        '+'  shift 3
        .  error


state 3
        expression : expression '+' . IDENTIFIER  (1)

        IDENTIFIER  shift 4
        .  error


state 4
        expression : expression '+' IDENTIFIER .  (1)

        .  reduce 1


4 terminals, 2 nonterminals
3 grammar rules, 5 states
```

y.tab.c

```
#ifndef lint
static char const
yyrcsid[] = "$FreeBSD: src/usr.bin/yacc/skeleton.c,v 1.28 2000/01/17
02:04:06 bde Exp $";
#endif
#include <stdlib.h>
#define YYBYACC 1
#define YYMAJOR 1
#define YYMINOR 9
#define YYLEX yylex()
#define YYEMPTY -1
#define yyclearin (yychar=(YYEMPTY))
#define yyerrok (yyerrflag=0)
#define YYRECOVERING() (yyerrflag!=0)
static int yygrowstack();
#define YYPREFIX "yy"
#define YYERRCODE 256
#define IDENTIFIER 257
const short yylhs[] = {                                        -1,
    0,    0,
};
const short yylen[] = {                                         2,
    3,    1,
};
const short yydefred[] = {                                      0,
    2,    0,    0,    1,
};
const short yydgoto[] = {                                       2,
};
const short yysindex[] = {                                   -257,
    0,  -42, -255,    0,
};
const short yyrindex[] = {                                      0,
    0,    0,    0,    0,
};
const short yygindex[] = {                                      0,
};
#define YYTABLESIZE 2
const short yytable[] = {                                       1,
    3,    4,
};
const short yycheck[] = {                                     257,
   43,  257,
};
#define YYFINAL 2
#ifndef YYDEBUG
#define YYDEBUG 0
#endif
#define YYMAXTOKEN 257
#if YYDEBUG
const char * const yyname[] = {
```

```
"end-of-
file",0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,
0,0,0,0,0,0,0,0,0,"'+'",0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,"IDENTIFIER",
};
const char * const yyrule[] = {
"$accept : expression",
"expression : expression '+' IDENTIFIER",
"expression : IDENTIFIER",
};
#endif
#ifndef YYSTYPE
typedef int YYSTYPE;
#endif
#if YYDEBUG
#include <stdio.h>
#endif
#ifdef YYSTACKSIZE
#undef YYMAXDEPTH
#define YYMAXDEPTH YYSTACKSIZE
#else
#ifdef YYMAXDEPTH
#define YYSTACKSIZE YYMAXDEPTH
#else
#define YYSTACKSIZE 10000
#define YYMAXDEPTH 10000
#endif
#endif
#define YYINITSTACKSIZE 200
int yydebug;
int yynerrs;
int yyerrflag;
int yychar;
short *yyssp;
YYSTYPE *yyvsp;
YYSTYPE yyval;
YYSTYPE yylval;
short *yyss;
short *yysslim;
YYSTYPE *yyvs;
int yystacksize;
/* allocate initial stack or double stack size, up to YYMAXDEPTH */
static int yygrowstack()
{
    int newsize, i;
    short *newss;
    YYSTYPE *newvs;
```

```
    if ((newsize = yystacksize) == 0)
        newsize = YYINITSTACKSIZE;
    else if (newsize >= YYMAXDEPTH)
        return -1;
    else if ((newsize *= 2) > YYMAXDEPTH)
        newsize = YYMAXDEPTH;
    i = yyssp - yyss;
    newss = yyss ? (short *)realloc(yyss, newsize * sizeof *newss) :
      (short *)malloc(newsize * sizeof *newss);
    if (newss == NULL)
        return -1;
    yyss = newss;
    yyssp = newss + i;
    newvs = yyvs ? (YYSTYPE *)realloc(yyvs, newsize * sizeof *newvs) :
      (YYSTYPE *)malloc(newsize * sizeof *newvs);
    if (newvs == NULL)
        return -1;
    yyvs = newvs;
    yyvsp = newvs + i;
    yystacksize = newsize;
    yysslim = yyss + newsize - 1;
    return 0;
}

#define YYABORT goto yyabort
#define YYREJECT goto yyabort
#define YYACCEPT goto yyaccept
#define YYERROR goto yyerrlab

#ifndef YYPARSE_PARAM
#if defined(__cplusplus) || __STDC__
#define YYPARSE_PARAM_ARG void
#define YYPARSE_PARAM_DECL
#else /* ! ANSI-C/C++ */
#define YYPARSE_PARAM_ARG
#define YYPARSE_PARAM_DECL
#endif      /* ANSI-C/C++ */
#else /* YYPARSE_PARAM */
#ifndef YYPARSE_PARAM_TYPE
#define YYPARSE_PARAM_TYPE void *
#endif
#if defined(__cplusplus) || __STDC__
#define YYPARSE_PARAM_ARG YYPARSE_PARAM_TYPE YYPARSE_PARAM
#define YYPARSE_PARAM_DECL
#else /* ! ANSI-C/C++ */
#define YYPARSE_PARAM_ARG YYPARSE_PARAM
#define YYPARSE_PARAM_DECL YYPARSE_PARAM_TYPE YYPARSE_PARAM;
#endif      /* ANSI-C/C++ */
#endif      /* ! YYPARSE_PARAM */

int
yyparse (YYPARSE_PARAM_ARG)
    YYPARSE_PARAM_DECL
{
    register int yym, yyn, yystate;
#if YYDEBUG
```

```
    register const char *yys;

    if ((yys = getenv("YYDEBUG")))
    {
        yyn = *yys;
        if (yyn >= '0' && yyn <= '9')
            yydebug = yyn - '0';
    }
#endif

    yynerrs = 0;
    yyerrflag = 0;
    yychar = (-1);

    if (yyss == NULL && yygrowstack()) goto yyoverflow;
    yyssp = yyss;
    yyvsp = yyvs;
    *yyssp = yystate = 0;

yyloop:
    if ((yyn = yydefred[yystate])) goto yyreduce;
    if (yychar < 0)
    {
        if ((yychar = yylex()) < 0) yychar = 0;
#if YYDEBUG
        if (yydebug)
        {
            yys = 0;
            if (yychar <= YYMAXTOKEN) yys = yyname[yychar];
            if (!yys) yys = "illegal-symbol";
            printf("%sdebug: state %d, reading %d (%s)\n",
                    YYPREFIX, yystate, yychar, yys);
        }
#endif
    }
    if ((yyn = yysindex[yystate]) && (yyn += yychar) >= 0 &&
            yyn <= YYTABLESIZE && yycheck[yyn] == yychar)
    {
#if YYDEBUG
        if (yydebug)
            printf("%sdebug: state %d, shifting to state %d\n",
                    YYPREFIX, yystate, yytable[yyn]);
#endif
        if (yyssp >= yysslim && yygrowstack())
        {
            goto yyoverflow;
        }
        *++yyssp = yystate = yytable[yyn];
        *++yyvsp = yylval;
        yychar = (-1);
        if (yyerrflag > 0)  --yyerrflag;
        goto yyloop;
    }
    if ((yyn = yyrindex[yystate]) && (yyn += yychar) >= 0 &&
            yyn <= YYTABLESIZE && yycheck[yyn] == yychar)
    {
        yyn = yytable[yyn];
```

```
        goto yyreduce;
    }
    if (yyerrflag) goto yyinrecovery;
#if defined(lint) || defined(__GNUC__)
    goto yynewerror;
#endif
yynewerror:
    yyerror("syntax error");
#if defined(lint) || defined(__GNUC__)
    goto yyerrlab;
#endif
yyerrlab:
    ++yynerrs;
yyinrecovery:
    if (yyerrflag < 3)
    {
        yyerrflag = 3;
        for (;;)
        {
            if ((yyn = yysindex[*yyssp]) && (yyn += YYERRCODE) >= 0 &&
                    yyn <= YYTABLESIZE && yycheck[yyn] == YYERRCODE)
            {
#if YYDEBUG
                if (yydebug)
                    printf("%sdebug: state %d, error recovery shifting\
 to state %d\n", YYPREFIX, *yyssp, yytable[yyn]);
#endif
                if (yyssp >= yysslim && yygrowstack())
                {
                    goto yyoverflow;
                }
                *++yyssp = yystate = yytable[yyn];
                *++yyvsp = yylval;
                goto yyloop;
            }
            else
            {
#if YYDEBUG
                if (yydebug)
                    printf("%sdebug: error recovery discarding state
%d\n",
                            YYPREFIX, *yyssp);
#endif
                if (yyssp <= yyss) goto yyabort;
                --yyssp;
                --yyvsp;
            }
        }
    }
    else
    {
        if (yychar == 0) goto yyabort;
#if YYDEBUG
        if (yydebug)
        {
            yys = 0;
            if (yychar <= YYMAXTOKEN) yys = yyname[yychar];
```

```
            if (!yys) yys = "illegal-symbol";
            printf("%sdebug: state %d, error recovery discards token %d
(%s)\n",
                    YYPREFIX, yystate, yychar, yys);
        }
#endif
        yychar = (-1);
        goto yyloop;
    }
yyreduce:
#if YYDEBUG
    if (yydebug)
        printf("%sdebug: state %d, reducing by rule %d (%s)\n",
                YYPREFIX, yystate, yyn, yyrule[yyn]);
#endif
    yym = yylen[yyn];
    yyval = yyvsp[1-yym];
    switch (yyn)
    {
    }
    yyssp -= yym;
    yystate = *yyssp;
    yyvsp -= yym;
    yym = yylhs[yyn];
    if (yystate == 0 && yym == 0)
    {
#if YYDEBUG
        if (yydebug)
            printf("%sdebug: after reduction, shifting from state 0 to\
 state %d\n", YYPREFIX, YYFINAL);
#endif
        yystate = YYFINAL;
        *++yyssp = YYFINAL;
        *++yyvsp = yyval;
        if (yychar < 0)
        {
            if ((yychar = yylex()) < 0) yychar = 0;
#if YYDEBUG
            if (yydebug)
            {
                yys = 0;
                if (yychar <= YYMAXTOKEN) yys = yyname[yychar];
                if (!yys) yys = "illegal-symbol";
                printf("%sdebug: state %d, reading %d (%s)\n",
                        YYPREFIX, YYFINAL, yychar, yys);
            }
#endif
        }
        if (yychar == 0) goto yyaccept;
        goto yyloop;
    }
    if ((yyn = yygindex[yym]) && (yyn += yystate) >= 0 &&
            yyn <= YYTABLESIZE && yycheck[yyn] == yystate)
        yystate = yytable[yyn];
    else
        yystate = yydgoto[yym];
#if YYDEBUG
```

```
    if (yydebug)
        printf("%sdebug: after reduction, shifting from state %d \
to state %d\n", YYPREFIX, *yyssp, yystate);
#endif
    if (yyssp >= yysslim && yygrowstack())
    {
        goto yyoverflow;
    }
    *++yyssp = yystate;
    *++yyvsp = yyval;
    goto yyloop;
yyoverflow:
    yyerror("yacc stack overflow");
yyabort:
    return (1);
yyaccept:
    return (0);
}
```

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”
```

Mixing definitions between yacc and lex

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 (or in a tightly coupled range).

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 whenever possible. It is also a quick way to check if you have unwanted, extra tokens.

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, lets 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 identifiy 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.

```
#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 10] src.c, line 10 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 */

%{
#define YYSTYPE double
%}

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

line
      :  /* empty */
      |  line expression '\n'
            { printf("%d\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 { yyerrok; }
           | 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: /* null */ | x y { yyerrok; } | x error
sequence	x: y { y }	x: y | x y { yyerrok; } | error | x error
list	x: y { T y }	x: y | x T y { yyerrok;} | error | x error | x error y {yyerrok;} | x T 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
- 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:

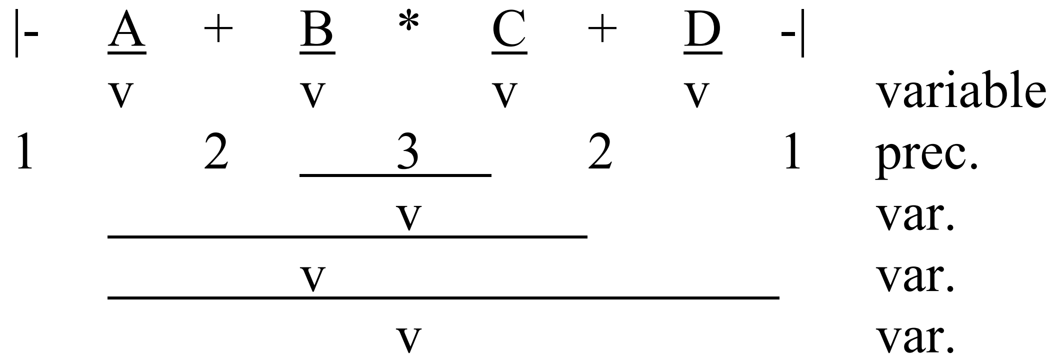
```
A + B * C + D
  2   3   2
```

Higher numbered operators are done first. Look for peaks:

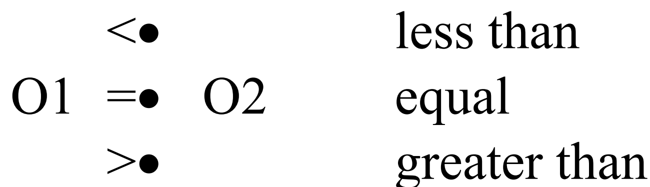
```
O1 v O2 v O3    (v = variable, O = operator)
P1 < P2 >= P3   (P = precedence)
```

Then replace “v O2 v” with “v” to get “O1 v O3”.

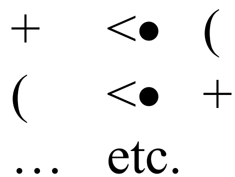
The process is like drawing brackets all with the same name:



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



We note that



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

A prec B if and only if f(A) prec g(B).

The table form of our grammar is

<table>
<tr><td></td><td></td><td colspan="8">Input symbol</td></tr>
<tr><td></td><td>Stack</td><td>|-</td><td>-|</td><td>;</td><td>=</td><td>+</td><td>*</td><td>(</td><td>)</td></tr>
<tr><td>0</td><td>|-</td><td></td><td>Stop</td><td><•</td><td><•</td><td><•</td><td><•</td><td><•</td><td></td></tr>
<tr><td>1</td><td>-|</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>2</td><td>;</td><td></td><td>•></td><td>•></td><td><•</td><td><•</td><td><•</td><td><•</td><td>•></td></tr>
<tr><td>3</td><td>=</td><td></td><td>•></td><td>•></td><td><•</td><td><•</td><td><•</td><td><•</td><td>•></td></tr>
<tr><td>4</td><td>+</td><td></td><td>•></td><td>•></td><td></td><td>•></td><td><•</td><td><•</td><td>•></td></tr>
<tr><td>5</td><td>*</td><td></td><td>•></td><td>•></td><td></td><td>•></td><td>•></td><td><•</td><td>•></td></tr>
<tr><td>6</td><td>(</td><td></td><td></td><td><•</td><td><•</td><td><•</td><td><•</td><td><•</td><td>=•</td></tr>
<tr><td>7</td><td>)</td><td></td><td>•></td><td>•></td><td></td><td>•></td><td>•></td><td></td><td>•></td></tr>
</table>

The functional form can be

<table>
<tr><th>a</th><th>f(a)</th><th>g(a)</th></tr>
<tr><td>|-</td><td>1</td><td></td></tr>
<tr><td>-|</td><td></td><td>1</td></tr>
<tr><td>;</td><td>4</td><td>3</td></tr>
<tr><td>=</td><td>4</td><td>5</td></tr>
<tr><td>+</td><td>7</td><td>6</td></tr>
<tr><td>*</td><td>9</td><td>8</td></tr>
<tr><td>(</td><td>2</td><td>11</td></tr>
<tr><td>)</td><td>12</td><td>2</td></tr>
</table>

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.,

```
<Name> ::= <Letter>                  (1)
         |  <Letter><Name>           (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).

```
<W> ::= <X> <Y> <Z>        (1)
      | <X>                (2)
      | <X> <Y> <M>        (3)
      | <F>                (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

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

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
}
```

<u>Early's Algorithm</u> (CACM, February, 1970)

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

<E> ::= <E> + <T> (1)
| <T> (2)
<T> ::= <T> * <F> (3)
| <F> (4)
<F> ::= a (5)
(1) (2) (3)

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

a	+	**a**
<E>::=**.**<E>+<T>	<T>::=<F>**.**	<E>::=<E>+**.**<T>
<E>::=**.**<T>	<F>::=a**.**	…
…	…	
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: <T> ::= <T> * . <F>
4-tuple: <3,3,,>

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

1. Predictor   Nonterminal <T> ::= <T> * . <F>
   We need to construct a <F>
   Add all productions about how to build a <F>
2. Scanner   Terminal <T> ::= <T> . * <F>
   We need a *
   Check against input
3. Completer Nothing <T> ::= <T> * <F>
   We have made a <T> 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
Unambigous	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*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*(#states))
Completer	Do "s&a" ~ 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).

Unambigous grammars:

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

Example: (Palindromes) <A> ::= x | x <A> x
Takes N^2 time to produce ~N brackets because ~(i/2) parses must be kept track of for the ith symbol.

Bounded Set Grammars:

No state bigger than some constant K.

Scanner and Predictor	time K
Completer	time K^2

⇨ $\leq (K+K^2)N$ time total

LR(k) Parsers

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

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

1. Closure (I, a set of items)
   If $A \rightarrow \alpha\mathbf{.}Bb$ and $B \rightarrow \mathbf{.}\gamma \mid \mathbf{.}\delta$
   then
   $B \rightarrow .\gamma$
   $B \rightarrow .\delta$
   are both added to Closure(I), where
   $\text{Closure}(I) = I \cup \{ \{B \rightarrow . \ldots\} \mid A \rightarrow \alpha.Bb \in I \}$.
2. Goto (I: set of items, X: symbol)
   For all $A \rightarrow \delta\mathbf{.}X\beta \in I$, define Goto as
   Closure $(A \rightarrow \alpha X\mathbf{.}\beta)$
   Intuitively, if I is the set of items valid for some prefix γ, then
   $\text{Goto}(I,X) = \{$ 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' → S
   where S is the old start symbol.
   Set C = Closure(S').
   While sensible,
      { For all I∈C and for any grammar symbol X,
        add Goto(I,X) to C }

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

Example:
S → aAcBe
A → Ab
A → b
B → d
(S' → S)

Items:

S' →	.S	A →	.Ab
	S.		A.b
S →	.aAcBe		Ab.
	a.AcBe		.b
	aA.cBe		b.
	aAc.Be	B →	.d
	aAcB.e		d.
	aAcBe.		

I_0:	$S' \rightarrow .S$	
	$S \rightarrow .aAcBe$	
I_1:	$S' \rightarrow S.$	Goto(I_0,S)
I_2:	$S \rightarrow a.AcBe$	Goto(I_1,a)
	$A \rightarrow .Ab$	
	$A \rightarrow .b$	
I_3:	$S \rightarrow aA.cBe$	Goto(I_2,A)
	$A \rightarrow b.$	
I_4:	$A \rightarrow b.$	Goto(I_2,b)
I_5:	$S \rightarrow aAc.Be$	Goto(I_3,c)
	$B \rightarrow .d$	
I_6:	$A \rightarrow Ab.$	Goto(I_3,b)
I_7:	$S \rightarrow aAcB.e$	Goto(I_5,B)
I_8:	$S \rightarrow aAcBe.$	Goto(I_7,e)
I_9:	$B \rightarrow d.$	Goto(I_5,d)

Example: <E’> ::= <E>
<E> ::= <E> + <T> | <T>
<T> ::= <T> * <F> | <F>
<F> ::= id | (<E>)

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.

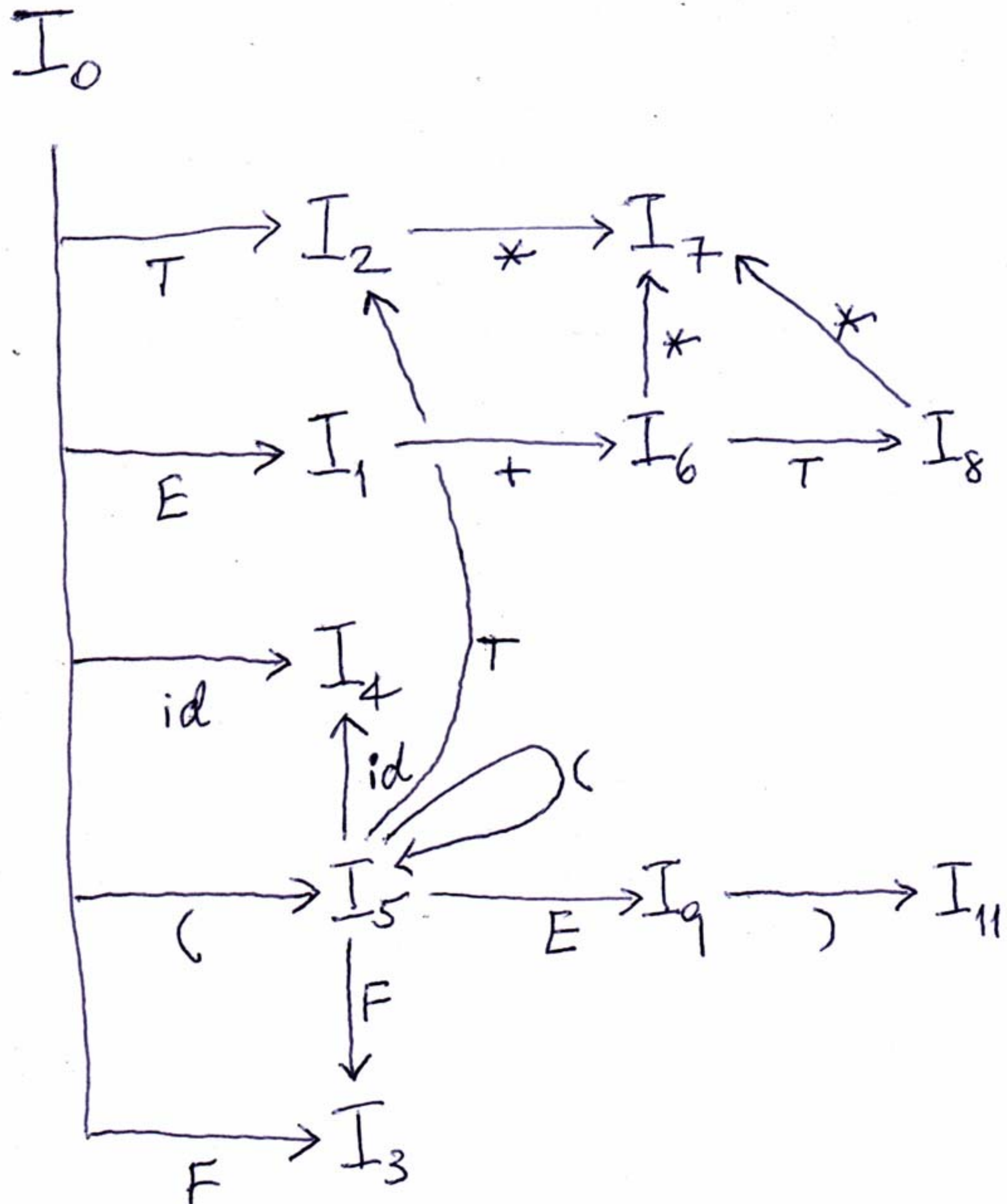
F ::= .id
id.
.(E)
(.E)
(E.)
(E).

See the transition diagram on p. 70 for motivation. The states here are a bit mystifying to just look at (i.e., a rabbit pulled out of a hat syndrome).

I_0:	E' →	.E	
	E →	.E+T	
	E →	.T	
	T →	.T*F	
	T →	.F	
	F →	.id	
	F →	.(E)	
I_1:	E' →	E.	Goto(I_0,E)
	E →	E.+T	
I_2:	E →	T.	Goto(I_0,T),Goto(I_5,T)
	T →	T.*F	
I_3:	T →	F.	Goto(I_0,F),Goto(I_5,F)
I_4:	F →	id.	Goto(I_0,id),Goto(I_5,id), Goto(I_6,id)

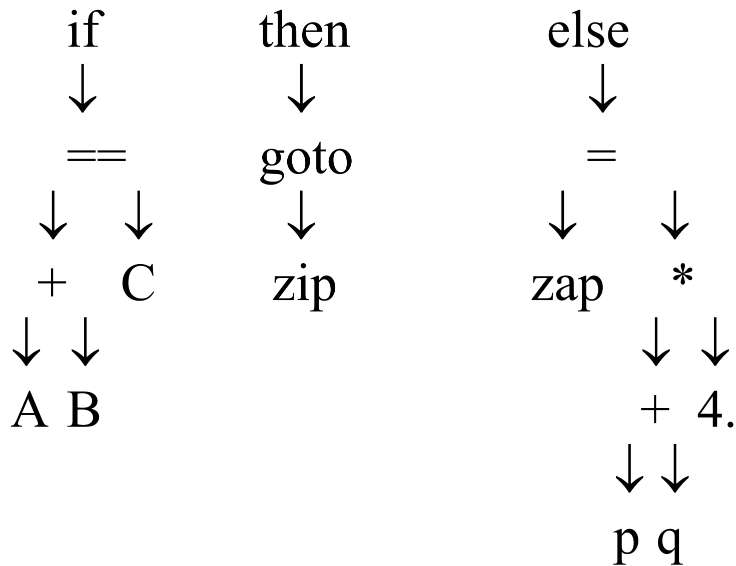
I_5:	$F \rightarrow$	(.E)	Goto(I_0,(),Goto(I_5,(),
	$E \rightarrow$	.E+T	Goto(I_6,()
	$E \rightarrow$	.T	
	$T \rightarrow$	.T*F	
	$T \rightarrow$	.F	
	$F \rightarrow$	.id	
	$F \rightarrow$	.(E)	
I_6:	$E \rightarrow$	E+.T	Goto(I_1,+)
	$T \rightarrow$	.T*F	
	$T \rightarrow$	.F	
	$F \rightarrow$	.id	
	$F \rightarrow$	.(E)	
I_7:	$E \rightarrow$	T*.F	Goto(I_2,*), Goto(I_6,*),
	$F \rightarrow$	.id	Goto(I_8,*)
	$F \rightarrow$	.(E)	
I_8:	$E \rightarrow$	E+T.	Goto(I_6,T)
I_9:	$F \rightarrow$	(E.)	Goto(I_9,E)
	$E \rightarrow$	E.+T	
I_{10}:	$T \rightarrow$	T*F.	Goto(I_7,F)
I_{11}:	$F \rightarrow$	(E).	Goto(I_9,))

Transition diagram



Tree representation

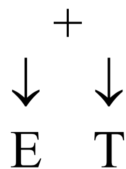
Example: if A+B == C
then goto zip
else zap = (p+q)*4.



When you do a reduction

$E \rightarrow E + T$

You produce



Tree branch and continue parsing.

Data structures can look similar to

```
struct ite {    int            op;
                struct cmp     *ite_comp;
                struct node    *ite_then;
                struct node    *ite_else; };

struct go {     int            op;
                struct sym     *go_sym; };

struct asgn {   int            op;
                struct sym     *a_sym;
                struct expr    *a_expr; };

struct conf {   int            op;
                float          cf_data; }
```

and so on…

Dynamic Allocation

ap = calloc(sizeof(z)), where

z is a structure
ap is a pointer

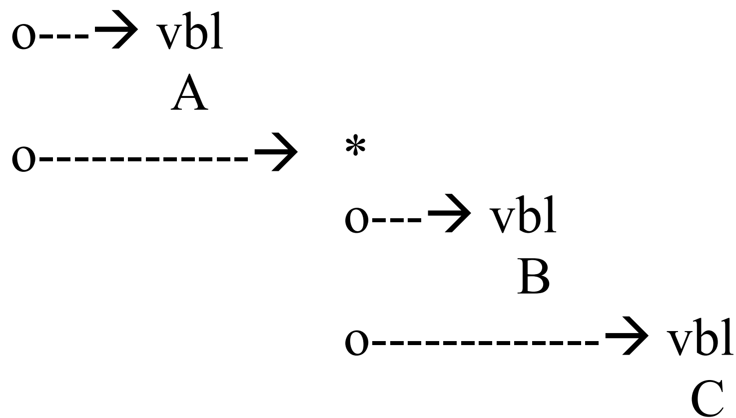
Example:

```
struct {  int   op;
          struct sym rvp };

struct polish {   int   what;
                  int   xx; };
switch( p→what ) {
      case vbl:   ap = calloc( sizeof(z) );
                  ap → op = vbl;
                  ap → vp = p→xx;
                  *sp++ = pp;
                  break;
      …
}
```

Example: $\quad$ Exp → Exp * Exp

```
struct arith { int   op;
               struct   arith   *O1;
               struct   arith   *O2;
             } zz;

switch( p→what ) {
    case mult:   ap = calloc( sizeof(zz) );
                 ap → ap = mult;
                 sp → O1 = *--sp;
                 ap → O2 = *--sp;
                 *sp++ = ap;
                 break;
        …
}
```

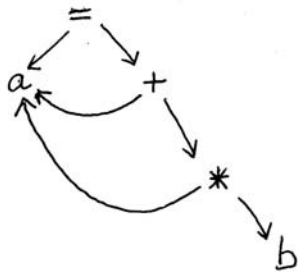
Consider A B C * = on the stack:

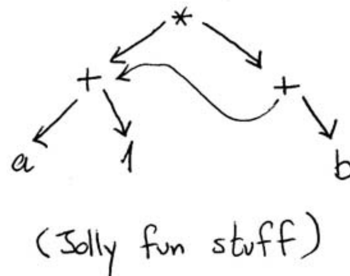


Directed Acyclic Graphs (dag)

DAgs are useful for expression evaluation in identifying common subexpressions. Like a tree, there is one node for every subexpression with children nodes being operands. However, a node can have multiple parents.

Examples: a = a + a * b (a+1)*(a+1+b)





Existing algorithms for trees can be modified to make a new node only if necessary and return pointer to existing node when possible. This is why structures (or records in arrays) are popular.

Backtracking in Parsing

Computer generated parsers spend an inordinate amount of time backtracking. Let's see why…

Example: Algol assignment statement.

```
<AssignStmt> ::= <Lval> := <AssignStmt>  (3)
              |  <Lval> := <Expr>         (4)
```

```
  A :=   B :=     C ;
Lval   Lval     Lval
       ______________
         AssignStmt
_____________________
     AssignStmt
```

So when you have A := B[I], there are two parses possible (oh, goodie):

```
1.  A :=      B[I]
   Lval       Lval
          AssignStmt

2.  A :=      B[I]
   Lval       Lval
              Expr
```

More inspection may be necessary to determine which one is correct at the input symbol point.

Top Down with Backup

<StList> ::= <StList> ; <AssignStmt>   (2)
           | <AssignStmt>   (1)

```
Procedure AssignStmt() {
    SI = I          (save I – input)
    SO = O;         (save O – output)
    If ( Lval() ) {
        If ( Input[I] == ':=' ) {
            I = I+1
            If ( Expr() ) {
                Out(3); Return 1;
            }
            If ( AssignStmt() ) {
                Out(4); Return 1
            }
        }
        I = SI; O = SO; Return 0
    }
}
```

There are pitfalls in this implementation, as we will see shortly. Order of checks counts!

```
Procedure StList() {
    If ( AssignStmt() ) {
        Out(1)
        While ( Input[I] == ';' ) {
            I = I+1; SaveO = O;
            If ( AssignStmt() ) {
                Out(2); Return 2
            }
            I = I – 1; O = SaveO
        }
        Return 1
    }
    Return 0
}
```

Problem with

```
  A :=    B :=     C ;
Lval    Expr     Lval
        -------------
  AssignStmt
-----------------------
```

is because we "Expr" first, find one, and return from AssignStmt. Then we do not find ";" so we end up with an error. The solution is to try AssignStmt before Expr in AssignStmt since a Lval can masquerade as an Expr.

Example: <S> ::= <B> a a .
<B> ::= b
| <B> a

This example is LR(2), but can be extended to any complexity:

```
b    a    a    a    a    .
B___
____B____
_______B_______________
           S
```

B does not need to know when to stop eating a's. We need to back up inside of the B program. In order to back up, we need to remember all untried alternatives. Recursive procedures are actually quite awkward in this case. Writing the grammar as a search tree helps, e.g.,

<Sum> ::= <Term> | <Term> ± <Sum>

instead as

```
<Sum> → <Term> → + → <Sum> → *1
                 ↓
                 − → <Sum> → *2
```

We can present a tree with either structures or arrays, e.g.,

Start[I]	Go to node for nonterminal I
Right[I]	Pointers
Down[I]	Pointers
Val[I]	Token at node
NT[I]	0 Terminal
	1 Nonterminal
	2 Leaf containing production no.

You must maintain a stack that gives a path through the tree

$$\langle S\rangle \rightarrow \langle B\rangle^{1} \rightarrow a^{2} \rightarrow a^{3} \rightarrow .^{4} \rightarrow *1^{5}$$

$$\langle B\rangle \rightarrow b^{6} \rightarrow *2^{7}$$

$$\downarrow$$

$$\langle B\rangle^{8} \rightarrow a^{9} \rightarrow *3^{10}$$

Notice that left recursion is ok here since it is tried last (this is an example of a well ordered tree).

baaaa. goes to (but drop <>'s from stack)

1	8	8	6	7	9	10	9	10	2	3	4	5
<B>	<B>	<B>	b	*2	a	*3	a	*3	a	a	.	*1

Bottom Up Backtracking

Same grammar, but write as

<B> a a . ==: <S> (1)
<B> a ==: <B> (2)
b ==: <B> (3)

Build a search tree as

<B>: $a^1 \rightarrow a^2 \rightarrow .^3 \rightarrow \langle S\rangle^4$ (1)
$\downarrow$
$\langle B\rangle^5$ (2)
b: $\langle B\rangle^6$ (3)

As with top down parsing, we have two stacks:

1. Path through the tree.
2. Currently unsatisfied goals.

The path stack is the output.

Palindrome grammar:

<B> ::= b (1)
<B> ::= a<B>a (2)

b: <B>^{1} (1)
a: a^2 → <B>^{3} → a^4 → <B>^{5} (2)

Goal stack: <B><B><B>

No alternative paths
No backtracking needed!

for legal input only

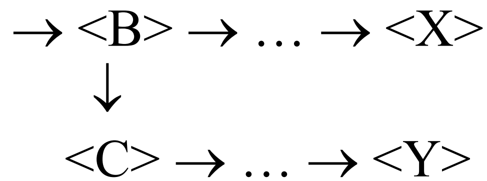
Do you stop or continue when you reach the current goal?

<Sum> : + → <Term> → <Sum>
↓
<Expr>

When you get to the rightmost <Sum> above and goal is <Sum>, do you go on?

Try to go on first.
Stop second.

However, for



and goal is <Z> and <Y> → → → <Z>
but <X> cannot → <Z>,

there is no point in trying the <B> branch.

LR systems can backtrack:

Shift-reduce conflict
Try one, backtrack and try another

Operator precedence

Table conflicts
(dubious)

Statistics on Errors

1. Roughly 67% of programs compiled are syntactically correct.
2. Of errors, most are punctuation, a few are operator/operand problems, a few are keyword/id errors, and less than 5% are anything else. Worst case analysis suggests that student programs have the highest error percentages.
3. ;&, problems are usual syntax errors in languages that use them. = intead of := or == are other biggies.
4. Compund errors in statements are about 15% of errors.

Type Checking

Static checks (done at compile time)
 Incompatible operator/operands
 Incompatible statement pairs

Flow control checks
 Statements which cause exit from a control structure (e.g., loop) must have a place to go to that is legal.
 Check if block of statements can be reached.

Uniqueness checks
 Declare a variable once per block
 Labels once per routine
 Labels once per case statement

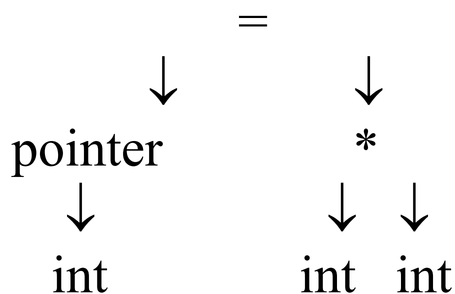
Name related checks
 Same name required twice (beginning and end of block)

Last three classes can be implemented easily as part of the parser. Type checking can be done as a separate step after the parse tree is generated or as part of parsing. Which is more efficient is language dependent (ADA – separate, Fortran/C/Pascal – not separate).

Type systems are collections of rules for assigning type expressions. Type expressions are basic elements of language types, e.g.,

integer, real, char,
array (domain)
function, records,
pointers

Type expressions usually follow parse tree or dag:



All checks are made *dynamically* (i.e. at run time). A *sound* type system is one where all type checking can be done statically by the compiler. A language is *strongly typed* if its compiler can guarantee that programs it accepts will execute without type errors. Type errors are another part of error recovery and should be handled gracefully.

The easiest way of doing type checking is to add associations to productions, e.g.,

$S \rightarrow$ if E then S_1

{ S.type = if (E.type == Boolean) then S_1.type
else error_type}

The types can be encoded is space is an issue.

Type Conversions

Example: x + i, x real, i integer – result is real.

Explicit systems require programmer intervention (C). *Implicit* systems require compiler intervention (Fortran).

Examples:

do I = 1 to n { x[i] = 1 }
do I = 1 to n { x[i] = 1.0 }

First form should be changed to the second form by the compiler (there is an enormous run time difference).

Polymorphic, built in functions: easily type checked.

Symbol Tables

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 useful to sort the entries in a symbol 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 with all inclusive information about a single object:

- An integer (*whatami*) describing what this is.
- The actual object, which can be (possibly a union structure) on 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 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 the dimensions. 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 do not even fall into this system and require a linked list of symbol table entries. At least this is simple to think about conceptually. ☺

Then there are special cases… ☹

Code Generation

You will generate very simple C code that you will pass through gcc. To be safe, you might want to check if your code works on a Linux system. I am willing to try only two systems (my laptop and the CS Lab machine salford).

Include file cs441-f02.h

The first thing in your generated code after a comment line identifying who you are

```
/* Itti compiler by Who, Done, and It */
```

should be a line with

```
#include < cs441-f02.h >
```

The Makefile will have a definition in it so that the file is found.

Do not modify this file 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 the 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 the class include file?

- A bunch of #define's.
- The global definitions for the memory types.
- A main program that initializes the stack register, calls yourmain(), and then returns.

When in doubt, read the file. If something is odd, ask for an explanation.

Memory

The memory is broken up into four objects:

- Integer Registers R[i], $0 \leq i < NR_Regs$
- Floating Point Registers F[i], $0 \leq i < NF_Regs$
- One Stack Register SR
- One Main Memory Mem[i], $0 \leq i < NF_Mem$

Note that R's are always *int*eger valued and F's are always *double* valued. Mem is also *int*eger valued, which makes addressing awkward for anything else e.g., classes, doubles, and strings.

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 equal to NF_Mem. The stack will grow down from the end of memory.

You should allocate memory from the beginning of memory when an allocate statement actually allocates an object. Unlike a real computer system, do not worry about freeing the memory or garbage collection. When memory runs out… oh, well.

Timing Considerations

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

```
ITTI_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 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 modulus	20
Double /	38

Things accumulate. For example,

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

It is OK to generate the timing in just this inefficient way. The main program in the class include file will print out the number of ITTI_Time ticks your code used.

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.

<u>Manipulate the Stack Register SR:</u>

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

Load (a register):

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

Store (a register):

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

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.

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 comparision operators.

Strictly no Mem's anywhere and no F's on the left hand side of the equal sign.

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;
return correct value on stack;

After the return, the stack may have to be manipulated in order to get return value. Be careful in your design.

Exit:

call Itti_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;

See return for comments on return values.

Printf/Scanf:

Do yourself a big favor and assume that no more than one easily referenced item is involved at a time.

I will not test your printf or scanf beyond class inspired examples that are trivial.

You should provide one or more examples that fit your compiler. Do not add any complexity whatsoever since it snowballs.

Constants:

These should be globally declared using a very set naming convention so that

1
1.001
"foobar"

get the same name no matter what. A dirty trick is to put them in an include file for compiling, which is included on the line after the class include file, e.g.,

```
#include "/tmp/.pid/myconstants.h"
```

All sorts of things can be legitimately swept under the rug here.

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;
(double)Mem[loc] = 0.0;
(double)Mem[R[2]] = 0.0;
```

No increment/decrement of doubles, but clearing OK.

Shift left or right (R registers only):

R[2] << integer constant;
R[2] >> integer constant;

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 comparision:

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).

Local Code Generation

Different types:

Arithmetic	A[I] = B + C * D
Control	X < Y -> P = Q
Function calls	F(X,Y+35)
Declarations	Real A[236]
Mixed arithmetic	Real + Int
Local optimization	A = A + 1

Arithmetic

A = (B+C) – (A*D)

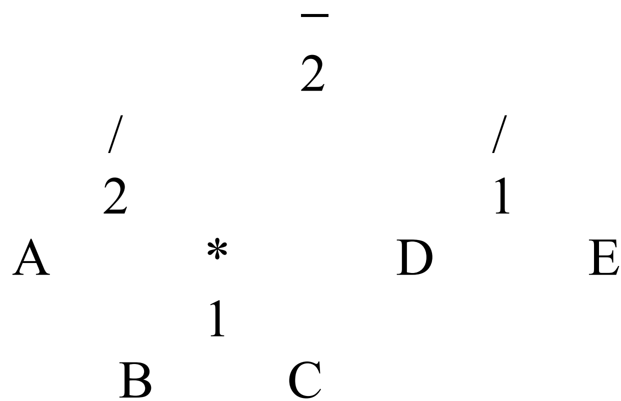
```
Mov     R1,A
Mov     R2,D
Mul     R2,R1    // double register output?
Mov     R0,B
Mov     R1,C
Add     R1,R0
Sub     R1,R2
Mov     A,R1
```

Two big questions:

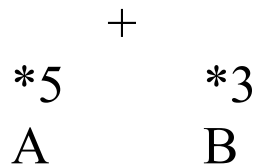
1. How many registers?
2. How do we assign them?

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, 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 op Y, one argument is also the result. However, consider A/(B/C):

```
Mov     R1,B
Div     R1,C
Mov     R2,A
Div     R2,R1
```

Note: Result argument takes a register if

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

<u>Assignment to Registers</u>

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 R	R0←A R0←R0–B	12
C	0 C R V		12
D	0 C D R V V		12
/	0 1 R R	R1←C R1←R1/D	2
*	0 R	R0←R0*R1	12

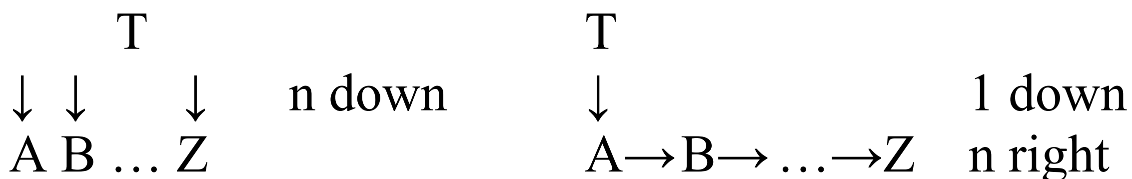
Further question: How to walk the tree:

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    −(SP),A
Mov    −(SP),B
Mov    −(SP),C
Mov    R1, (SP)+
Mul    R1, (SP)+
Mov    −(SP),R1
Add    (SP),(SP)+
```

The switch cases of interest can be written like

```
Case Var:    Write(" Mov -(SP),", T.Val)

Case *:      Gen(A)
             Gen(B)
             Write(" Mov R1,(SP)+")
             Write(" Mul R1,(SP)+")
             Write(" Mov –(SP),R1")

Case +:      Gen(A)
             Gen(B)
             Write(" Add (SP),(SP)+")
```

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 X < Y then P = Q else N = M
(A = X < Y, B = P = Q, C = N = M)

```
Case ITE:   I = Gen(A)
            T = NextLabel()
            Write(" ", I, " ", T)
            Gen(B)
            S = NextLabel()
            Write(" Br ",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.

<u>What about register allocation? (top down approach)</u>

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 = Gen(A) The order is determined
PB = Gen(B) by # of registers used.
If (A not a register) { Put in a register }
Write(" Add", PB→Des, ",",PA→Des)
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 = Gen(B):

B	*Type*	*Value*	*Comment*
A	Var	A	
35	Const	#35	
A[35]	Var	A(R3)	reg. containinng 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( " Mov R", R, ",", P→Addr )
        P→Type = 'R'
        P→Addr = 'R',R
        Return
        }
```

Now there are two easy ways of doing PB = Gen(B):

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

In both cases the user may (must) have to explicitly free storage. Further, the freeing should be done to avoid memory leak problems in general.

```
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(" Add", 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.

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(~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 contructs as well (do, while, repeat, break, next, …)… even to Itti.

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 (sadly):

Size

int	16, 32, or 64 bits
long	32 or 64 bits
char	8 or 16 bits
float	32, 38, 64, or 80 bits
double	56, 64, or 128 bits

Scope (what a program knows about a variable):
- All functions
  - external
- One function
  - static, auto, register

Storage class (where the variable is located)
- Registers
  - General purpose
  - Floating point
- On the stack
- In memory
- In cache
- Constant
- Heap (dynamic memory)

The Stack

A stack or frame register is maintained. It may be a separate, special purpose register or one of the general purpose registers may be designated.

Stacks grow up or down in memory (depends on which of two patents people used to use in the *old days*). A frame inside the stack is a useful concept when thinking about what local variables are on the stack and actively accessible.

Example:

```
begin     int   A, X[2];       // Block 1
          …
          begin     int B, Y[A]   // Block 2
                    …
          end
end
```

```
-------------
A     (#4)          Frame for block 1
X[1]
X[2]
-------------
B                   Frame for block 2
Y[1]
Y[2]
Y[3}
Y[4]
-------------
                    Stack pointer
```

Of course there is a famous pointer problem:

```
begin int*p, v;
     begin int A[100];
          p = &A;
     end
     v = *p;   // storage for A has gone away
end
```

Languages like Algol68, Pascal, Java, … go to incredible lengths to detect this error at compile time. They also know what a pointer points at during compile time.

C and Fortran: this is a user problem – compiler allows anything.

<u>The Heap</u>

C:
```
int *p;
p = malloc(100);
```
allocates 100 bytes on the heap.
```
free(p);
```
frees the heap block whose 1st byte is pointed to by p. A good heap system checks if p points to something known or not.

Chapter 2 of volume 1 of Knuth's treatises contains a collection of algorithms for dynamic storage allocation.

Fortran-77: Everything that is not in named Common is on the heap unless a SAVE statement is applied. (This was mostly left unimplemented after almost uniform howls from the user community in late 1977 through 1978.)

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,

```
               A
               ↓
              *A
               ↓
edge    { A[0] →  ( *A[0][0], *A[0][1] )
vectors{ A[1] →  ( *A[1][0], *A[1][1] )
        { A[2] →  ( *A[2][0], *A[2][1] )
```

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

```
Mov     R1,@A       // *A
Add     R1,I
Mov     R1,@R1      // *A[I]
Add     R1,J
Mov     R1,@R1      // *A[I][J]
```

However, edges require extra storage for the edge vectors.

Another possibility is to use storage by rows.

```
 A
 ↓
*A →   *A[0][0]
       *A[0][1]
       …
       *A[2][1]
```

Now the code is

```
Mov   R0,@A      // R0 = *A
Mov   R1,I
Mul   R1,n       // n = #bytes in last dim.
Add   R0,R1      // R0 = *A[I]
Mov   R1,J
Mul   R1,n       // could also be a shift
Add   R0,R1
Mov   R1,@R0     // R0 = *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]

```
R1 = (&A) + (7*5*n)*I     // & A[I]
R1 = R1 + (5*n)*J         // &A[I][J]
R1 = R1 + n*K             // &A[I][J][K]
R1 = *R1                  // 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.

<u>Structures</u> (and classes)

These can be addressed similarly to arrays.

		Offset	Bytes
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 to 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)

   ```
   Mov R5,&R
   Goto F
   R: address of A
      address of B
      next instruction
   ```

   Get address of A in F with Mov R1,(R5)

   Get value of A in F with Mov R1,@(R5)

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[0] = &R;
F[1] = X;
F[2] = Y;
Goto 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 sizeC
2. converting floating point bumber to int in a store operation (double → float → 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
$b = a - d$	$\Rightarrow$	$b = a - d$	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 it to

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

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

$x = x \pm 0 \quad \rightarrow \quad$ no work
$x = x * 1 \quad \rightarrow \quad$ 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 lable is assumed to be that value.

Notes:

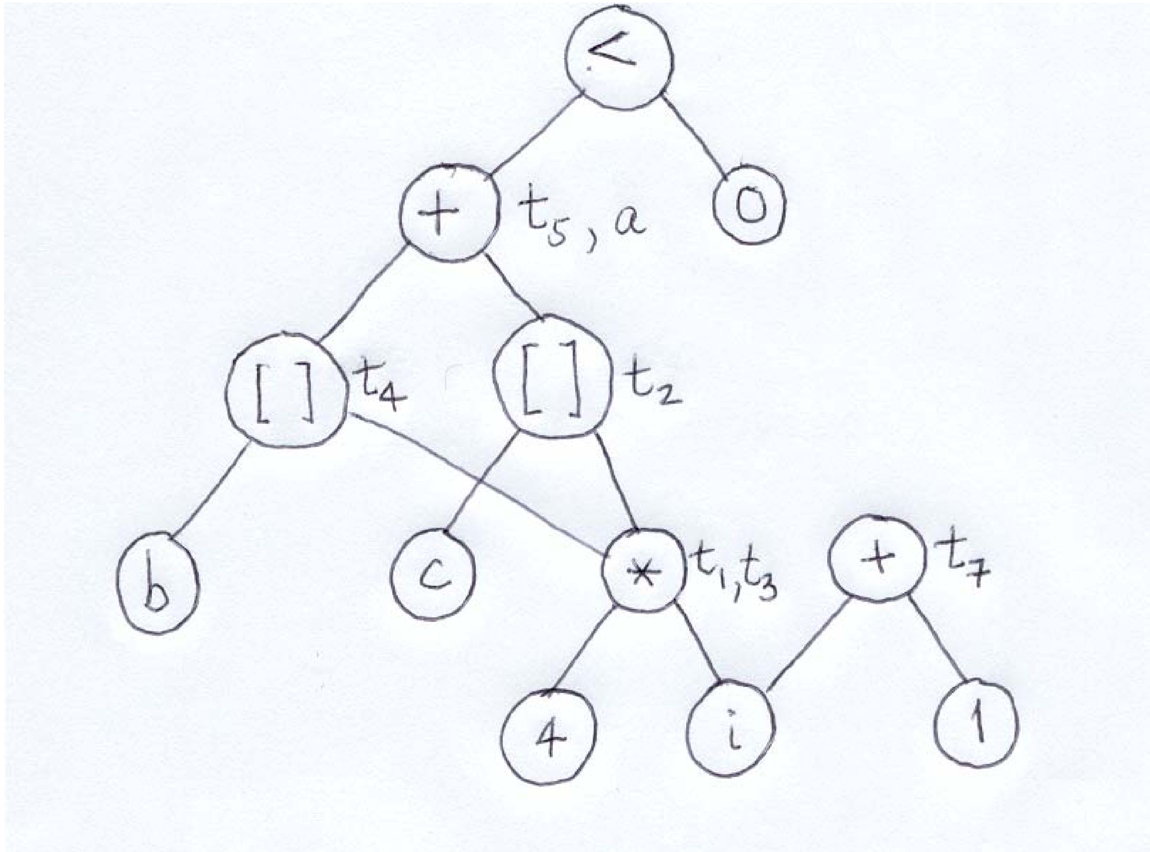
1. You have to have a DAG per basic block.
2. This is not related to the flow graph, but complementary.
3. If there is no common subexpression in a basic block, then we normally get a tree, not a DAG.

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 op z (this include if $y < z$ goto blah)

ii. x = op z

iii. x = y

Algorithm: For each statement in a basic block,

1. If node(y) is empty, then create a leaf y and let node(y) be this node. For case (i), repeat for node(z).
2. Case (i): Determine if node labeled op exists with left child node(y) and right child 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 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. The 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
int * 2n	→ shift left by n bits
float / constant	→ float * (*1/constant*)
…	→ the list is almost endless

Next Use Information

This is used in eliminating code that does useless computations or stores inside a basic block. It is also useful for global data flow analysis.

Start at the *end* of a block and scan *backwards*. For each name X in a 3-address statement, record in the symbol table whether or not it has a next use inside this block and if not, whether or not it is live on exit from the block.

If we are not doing global optimization, then all temporaries are dead on exit from each basic block. Any X in X = Y op Z that has no next use in the block and is a temporary causes elimination of the statement. For global optimization, we can look at data flow analysis to see if we can eliminate code.

Global Optimization

Most local optimization transformation can be extended to the global case. However, they should be applied locally each basic block first.

Some examples:

1. Common subexpression elimination now requires global data flow analysis.
2. Copy propogation

$x = t_1$		$x = t_1$
$a[t_2] = t_3$	$\Rightarrow$	$a[t_2] = t_3$
$a[t_4] = x$		$a[t_4] = t_1$
branch…		branch…

   Now we might be able to eliminate $x = t_1$ statement completely.
3. Dead code elimination

   if (false) { … }

   Evaluate the expressions to get a single constant if possible. This can eliminate more code.
4. Loop optimization is a family of optimizations…

Global Loop Optimization

Code motion

Find expressions that do not change inside a loop and move the code *before* the loop.

```
while ( i < n – 1 ) …
```

becomes

```
t1 = n – 1
while ( i < t1 ) …
```

This only works when the loop has a unique entry node. The amount of savings can be large as the number of loops nested grows. In 3D simulation codes have 3+ nested loops. Sometimes it is more like 6+ or 8+. Dumb code sometimes makes the code more readable (and hence, easier to maintain). The writer blindly assumes the compiler will clean up the sloppily written mess.

Induction variables

In loops, multiple variables may be in lockstep, e.g.,

```
do I = 1,n { a[i] = 0 }
```

produces

B1: $i = 1$
$t_1 = 4$
B2: $a[t_1] = 0$
$t_1 = t_1 + 4$
$i = i + 1$
if $i < n$ goto B2

Loop optimization can be used in conjunction with global data flow analysis to eliminate I in the last example and change the end of B2 to just

if $t_1 < t_2$ goto B2

If i is a dead variable at the end of the loop, more code can be eliminated. One thing to watch out for is language standards that do not allow user variable elimination inside of loops. This can be bypassed sometimes by precomputing the final value and storing that somewhat (before or after the loop).

Optimization of loops proceeds from the innermost loop to the outermost loop. It is frequently useful to create a *pre-header basic block* for any loop that is initially empty, but may have code moved into it.

For itti, the pre-header is natural since the language already requires support for a pre-header. However, you need to *another pre-header* for what is pulled out of the loop. Declared variable that have to be dynamically allocated are prime suspects to move into the extra pre-header.

Global Data Flow Analysis

This is based on data flow information accumulated by the compiler. For example, a data equation is

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

which means *the information at the end of statement S is either generated within S or enters at the beginning and is not killed as control flows through the statement*. We can develop similar concepts egen and ekill for expressions in a block. There are two main ways of solving equations: iterative and interval analysis.

Notes:

1. How gen and kill are defined is somewhat loose and depends on the desired results. In fact, in is a function of out in some applications.
2. In gen, data flow equations are defined within basic blocks.
3. Procedure calls, pointer assignments, and array variable assignments require finesse since they sometimes mess up data flow equations.

Reaching definitions: This is any statement that can assign a value to a variable (e.g., X). This includes, but is not exhaustive:

- X = …
- call foo(X), where foo might modify X.
- *p = …, where p → X is a possibility.

The latter is particularly problematic since without concrete information about what p points to, we have to assume it can point to anything.

Information is frequently stored in vectors. Space is saved by storing only what is absolutely necessary, e.g., register allocation optimization rules, frequency of use, etc.

Use-Definition (UD) chains are convenient to have. The contents of a UD chain for a variable X in a block B are

- If preceded by no unambiguous use of X, then just definitions in B that are definitions of X.
- If preceded by unambiguous use of X, then just the last definition in B.
- If ambiguous definitions of X, then all of these such that no unambiguous definition of X lies between it and use of X.

Iterative Solution of Data Flow Equations

Reaching definitions

Assume for each block B we have computed gen[B] and kill[B]. Then

$$in[B] = \cup\ out[P], \text{ where } P \in \text{prediction of } B$$

and

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

This can be computed using

```
Reach_Defn(flow graph, gen, kill) {
      out[B] = gen[B], ∀B
      repeat {
            for each block B {
                  in[B] = ∪ out[P], ∀pred. P
                  out[B] = gen[B] ∪(in-kill)
                  }
            } until no out[B] changes
      }
```

This algorithm propagates definitions as far as possible until they are killed. It simulates all possible

executions of the program. Under proper conditions, the repeat loop is executed ~5 times on normal user programs. Without care, a lot of memory is used in this algorithm.

Available expressions

Statements like Y+Z and X=Y+Z are computed using the same algorithm as before with egen and ekill substituted for gen and kill.

Defn-Use chains

are similar to…

Live variable analysis

Now define

- in[B] = {vars} live on entrance to B.
- out[B] = {vars} live on exit from B.
- def[B] = {vars} definitely assigned values in B prior to any use of that variable in B.
- use[B] = {vars} used in B before any definition of that variable.

The algorithm is similar to before:

```
Live_Var_Anal(flow graph, def, use) {
    in[B] = ∅, ∀B
    repeat {
        for each block B {
            out[B] = ∪in[S], S a successor in B
            in[B] = use[B] ∪(out[B]-def[B])
            }
        } until no in[B] changes
    }
```

U-D chain

Reaching definitions information is stored in a list. For each use of a variable, the list contains all of its definitions that reach that use.

D-U chain

List all possible uses of a definition.

Note: Many of these techniques were developed assuming structured programming was in use and that blocks would be small and well defined. 150K lines of code in a subroutine tends to break almost all of these techniques.

Code Improving Transformations

Several transformations are usually applied together. These include

- Elimination of global common subexpressions.
- Copy propagation.
- Detection of loop invariants.
- Code motion.
- Elimination of induction variables.

These techniques should be applied iteratively to maximize the gains.

```
Glob_Com_Subexpr_Elim(flow graph, avail
expr info)
    For every statement S: X=Y+Z such that
    Y+Z is available at the beginning of S and Y
    and Z are not defined prior to S in a basic
    block with S
      a) Follow the flow graph backwards to
         blocks that compute Y+Z (stop search
         when one is found).
      b) Create a new name U.
      c) Replace all W=Y+Z found in a) by
             U=Y+Z; W=U
      d) Replace S by X=U
```

Some comments are in order:

1. Only do this procedure for relevant Y+Z, not all.
2. Without copy propagation optimization, this algorithm produces worse code.
3. Several iterations of all optimization will catch

   A = X+Y; B = A*Z

   versus

   C = X+Y; D = C*Z

   (catching that B=D is not so easy).

Copy Propagation

- cgen[B] = { copies gen in B }
- ckill[B] = { copies killed in B }
- out[B] = cgen[B] $\cup$(in[B]-ckill[B])
- in[B] = $\emptyset$ if B = B_1 (initial block)
  in[B] = $\cap$out[P] otherwise, P a prediction of B

This is solved using the available expression algorithm

Copy_prop(…)
  For every S: X=Y,
   a) Determine uses of X that are reached by S.
   b) Determine whether for every use of X found in a),
      S$\in$cin[B] (B for particular use)
     and no definitions for X or Y occur prior to this use in B.
   c) If S satisfies b), remove S and replace all uses of X (found in a)) by Y.

Detecting Loop Invariant Components

```
Detect_loop_invar(basic blocks forming loop L) {
    For every statement in L {
        Mark invariant any statement whose
        operands are all constants or have all their
        reaching definitions outside of L. }
    Repeat {
        Mark invariant any unmarked statement
        whose operands are all
          a) Constants
          b) Have their reaching definitions outside
             of L, or
          c) Have exactly one reaching definition
             that is a statement in L marked
             invariant.
        } until no new statements are marked
        invariant
    }
```

Note that there is a heavy use of D-U chains here. Lisp-like constructs are really useful.

Code Motion

A block dominates all exits from a loop if that block must be executed before exit from the loop can occur on any pass through the loop. This can be determined by examining the flow graph for the loop.

```
Code_motion(loop L, UD chains, dominator info) {
   1. detect_loop_invar(L)
   2. for every S defining X found in 1, check
        a. that it is in a block dominating all exits of L
        b. X is not defined elsewhere in L
        c. All uses of X in L can only be reached by
           the definition of X in S.
   3. Move (without disturbing order) every statement
      S from 1 satisfying 2 to a pre-header basic block.
```

Note that condition 2a can be relaxed to include X's not used after exiting the loop.