Compilers (CS30003)

Lecture 19-20

Pralay Mitra

Autumn 2023-24

Handling Arithmetic Expression
A calculator grammar

1:	L	$\rightarrow$	$L\ S\ \backslash n$
2:	L	$\rightarrow$	$S\ \backslash n$
3:	S	$\rightarrow$	**id** $= E$
4:	E	$\rightarrow$	$E + E$
5:	E	$\rightarrow$	$E - E$
6:	E	$\rightarrow$	$E * E$
7:	E	$\rightarrow$	E / E
8:	E	$\rightarrow$	(E)
9:	E	$\rightarrow$	$- E$
10:	E	$\rightarrow$	**num**
11:	E	$\rightarrow$	**id**

Attributes and expression

- *E.loc*
  - Location to store the value of the expression
  - An entry to Symbol table
- *id.loc*
  - Location to store the value of the identifier *id*
  - An entry to Symbol table
- Num.val
  - Value of numeric constant

Auxiliary method for translation

- *gentemp()*
  - Generate a new temporary
  - Make an entry to Symbol Table
  - Return the pointer to the Symbol Table entry
- *emit(result, arg1,op,arg2)*
  - Spit a three address code:
    - Case 1 (binary operator): *result* = *arg1 op arg2*
    - Case 2 (unary operator): one *arg* is missing
    - Case 3 (copy instruction): one *arg* and op is missing

Expression grammar with Action

	Production Rule			Action
1:	L	$\rightarrow$	$L\ S\ \backslash n$	{}
2:	L	$\rightarrow$	$S\ \backslash n$	{}
3:	S	$\rightarrow$	**id** $= E$	{ *emit(id.loc=E.loc;* }
4:	E	$\rightarrow$	$E + E$	{ *E.loc=gentemp();* emit($E.loc=E_1.loc+E_2.loc$); }
5:	E	$\rightarrow$	$E - E$	{ *E.loc=gentemp();* emit($E.loc=E_1.loc-E_2.loc$); }
6:	E	$\rightarrow$	$E * E$	{ *E.loc=gentemp();* emit($E.loc=E_1.loc*E_2.loc$); }
7:	E	$\rightarrow$	E / E	{ *E.loc=gentemp();* emit($E.loc=E_1.loc/E_2.loc$); }
8:	E	$\rightarrow$	(E)	{ $E.loc=E_1.loc$; }
9:	E	$\rightarrow$	$- E$	{ *E.loc=gentemp();* emit($E.loc= -E_1.loc$); }
10:	E	$\rightarrow$	**num**	{ *E.loc=gentemp();* emit(*E.loc=num.val*); }
11:	E	$\rightarrow$	**id**	{ *E.loc=id.loc;* }

Translation with immediate spitting

- TAC are emitted as soon as they are formed.

```
%{
#include <string.h>
#include <iostream>
#include "parser.h"
extern int yylex();
void yyerror(const char *s);
#define NSYMS 20 /* max # of symbols */
symboltable symtab[NSYMS];
%}

%union {
    int intval;
    struct symtab *symp;
}

%token <symp> NAME
%token <intval> NUMBER

%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%type <symp> expression
%%

stmt_list: statement '\n'
         | stmt_list statement '\n'
         ;
```

```
statement: NAME '=' expression
    { emit($1->name, $3->name); }
        ;

expression: expression '+' expression
    { $$ = gentemp();
      emit($$->name, $1->name, '+', $3->name); }
        | expression '-' expression
    { $$ = gentemp();
      emit($$->name, $1->name, '-', $3->name); }
        | expression '*' expression
    { $$ = gentemp();
      emit($$->name, $1->name, '*', $3->name); }
        | expression '/' expression
    { $$ = gentemp();
      emit($$->name, $1->name, '/', $3->name); }
        | '(' expression ')'
    { $$ = $2; }
        | '-' expression %prec UMINUS
    { $$ = gentemp();
      emit($$->name, $2->name, '-');  }
        | NAME { $$ = $1; }
        | NUMBER
    { $$ = gentemp();
      printf("\t%s = %d\n", $$->name, $1); }
        ;
%%
```

```
/* Look-up Symbol Table */
symboltable *symlook(char *s) {
    char *p;
    struct symtab *sp;
    for(sp = symtab;
        sp < &symtab[NSYMS]; sp++) {
        /* is it already here? */
        if (sp->name &&
           !strcmp(sp->name, s))
            return sp;
        if (!sp->name) {
        /* is it free */
            sp->name = strdup(s);
            return sp;
        }
        /* otherwise continue to next */
    }
    yyerror("Too many symbols");
    exit(1); /* cannot continue */
} /* symlook */

/* Generate temporary variable */
symboltable *gentemp() {
    static int c = 0; /* Temp counter */
    char str[10]; /* Temp name */
    /* Generate temp name */
    sprintf(str, "t%02d", c++);
    /* Add temporary to symtab */
    return symlook(str);
}
```

```
/* Output 3-address codes */
void emit(char *s1, char *s2, char c, char *s3)
{
    if (s3)
        /* Assignment with Binary operator */
        printf("\t%s = %s %c %s\n",s1, s2, c, s3);
    else
        if (c)
            /* Assignment with Unary operator */
            printf("\t%s = %c %s\n",s1, c, s2);
        else
            /* Simple Assignment */
            printf("\t%s = %s\n",s1, s2);
}

void yyerror(const char *s) {
    std::cout << s << std::endl;
}

int main() {
    yyparse();
}
```

```
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers know about them. */
   enum yytokentype {
     NAME = 258,
     NUMBER = 259,
     UMINUS = 260
   };
#endif
/* Tokens. */
#define NAME 258
#define NUMBER 259
#define UMINUS 260

#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE {
#line 11 "calc.y" /* Line 2068 of yacc.c  */

    int intval;
    struct symtab *symp;

#line 67 "y.tab.h" /* Line 2068 of yacc.c  */
} YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
#endif

extern YYSTYPE yylval;
```

Translation with immediate spitting

```
#ifndef __PARSER_H
#define __PARSER_H

/* Symbol Table Entry */
typedef struct symtab {
    char *name;
    int value;
} symboltable;

/* Look-up Symbol Table */
symboltable *symlook(char *);

/* Generate temporary variable */
symboltable *gentemp();

/* Output 3-address codes */
/* if s3 != 0 ==> Assignment with Binary operator */
/* if s3 == 0 && c != 0 ==> Assignment with Unary operator */
/* if s3 == 0 && c == 0 ==> Simple Assignment */
void emit(char *s1, char *s2, char c = 0, char *s3 = 0);

#endif // __PARSER_H
```

```
%{
#include <math.h>
#include "y.tab.h"
#include "parser.h"
%}

ID          [A-Za-z][A-Za-z0-9]*

%%
[0-9]+      {
              yylval.intval = atoi(yytext);
              return NUMBER;
            }

[ \t]       ;  /* ignore white space */


{ID}        { /* return symbol pointer */
              yylval.symp = symlook(yytext);
              return NAME;
            }

"$"         { return 0; /* end of input */ }

\n|.        return yytext[0];
%%
```

Sample run

```
$ ./a.out
a = 2 + 3 * 4
    t00 = 2
    t01 = 3
    t02 = 4
    t03 = t01 * t02
    t04 = t00 + t03
    a = t04
b = (a + 5) / 6
    t05 = 5
    t06 = a + t05
    t07 = 6
    t08 = t06 / t07
    b = t08
c = (a + b) * (a - b) * -1
    t09 = a + b
    t10 = a - b
    t11 = t09 * t10
    t12 = 1
    t13 = - t12
    t14 = t11 * t13
    c = t14
$
```

Translation with lazy spitting

- Intermediate TAC are formed as quad and stored in an array. Spitting is done at the end of the output to facilitate later optimization.

```
%{
#include <string.h>
#include <iostream>
#include "parser.h"
extern int yylex();
void yyerror(const char *s);
#define NSYMS 20 /* max # of symbols */
symboltable symtab[NSYMS];
quad *qArray[NSYMS]; /* Store of Quads */
int quadPtr = 0; /* Index of next quad */
%}

%union {
    int intval;
    struct symtab *symp;
}

%token <symp> NAME
%token <intval> NUMBER

%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%type <symp> expression
%%

start: statement_list
    { for(int i = 0; i < quadPtr; i++)
          qArray[i]->print(); }
```

```
statement_list: statement '\n'
              | statement_list statement '\n'
              ;
statement: NAME '=' expression
  { qArray[quadPtr++] =
    new quad(COPY, $1->name, $3->name); }
         ;
expression: expression '+' expression
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(PLUS, $$->name, $1->name, $3->name); }
          | expression '-' expression
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(MINUS, $$->name, $1->name, $3->name); }
          | expression '*' expression
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(MULT, $$->name, $1->name, $3->name); }
          | expression '/' expression
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(DIV, $$->name, $1->name, $3->name); }
          | '(' expression ')'      { $$ = $2; }
          | '-' expression %prec UMINUS
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(UNARYMINUS, $$->name, $2->name); }
          | NAME                    { $$ = $1; }
          | NUMBER
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(COPY, $$->name, $1); }
          ;
%%
```

```
/* Look-up Symbol Table */
symboltable *symlook(char *s) {
    char *p;
    struct symtab *sp;
    for(sp = symtab;
        sp < &symtab[NSYMS]; sp++) {
        /* is it already here? */
        if (sp->name &&
           !strcmp(sp->name, s))
            return sp;
        if (!sp->name) {
        /* is it free */
            sp->name = strdup(s);
            return sp;
        }
        /* otherwise continue to next */
    }
    yyerror("Too many symbols");
    exit(1); /* cannot continue */
} /* symlook */

/* Generate temporary variable */
symboltable *gentemp() {
    static int c = 0; /* Temp counter */
    char str[10]; /* Temp name */
    /* Generate temp name */
    sprintf(str, "t%02d", c++);
    /* Add temporary to symtab */
    return symlook(str);
}
```

```
void yyerror(const char *s) {
    std::cout << s << std::endl;
}

int main() {
    yyparse();
}
```

Representation

- Quad is with the following fields

1. opcodeType op; // binary or unary or copy operation

/* use decimal form as string for numeric constant */

2. char *arg1; // argument 1
3. char *arg2; // argument 2
4. char *result; // result

```
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers know about them. */
   enum yytokentype {
     NAME = 258,
     NUMBER = 259,
     UMINUS = 260
   };
#endif
/* Tokens. */
#define NAME 258
#define NUMBER 259
#define UMINUS 260

#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE {
#line 13 "calc.y" /* Line 2068 of yacc.c  */

int intval;
struct symtab *symp;

#line 67 "y.tab.h" /* Line 2068 of yacc.c  */
} YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
#endif

extern YYSTYPE yylval;
```

```
#ifndef __PARSER_H
#define __PARSER_H

#include<stdio.h>

/* Symbol Table Entry */
typedef struct symtab {
char *name;
int value;
}symboltable;

/* Look-up Symbol Table */
symboltable *symlook(char *);

/* Generate temporary variable */
symboltable *gentemp();

typedef enum {
    PLUS = 1,
    MINUS,
    MULT,
    DIV,
    UNARYMINUS,
    COPY,
} opcodeType;
```

```
class quad {
    opcodeType op;
    char *result, *arg1, *arg2;
public:
    quad(opcodeType op1, char *s1, char *s2, char *s3=0):
        op(op1), result(s1), arg1(s2), arg2(s3) { }

    quad(opcodeType op1, char *s, int num):
        op(op1), result(s1), arg1(0), arg2(0)
    {
        arg1 = new char[15];
        sprintf(arg1, "%d", num);
    }

    void print() {
        if ((op <= DIV) && (op >= PLUS)) { // Binary Op
            printf("%s = %s ",result, arg1);
            switch (op) {
                case PLUS: printf("+"); break;
                case MINUS: printf("-"); break;
                case MULT: printf("*"); break;
                case DIV: printf("/"); break;
            }
            printf(" %s\n",arg2);
        }
        else
            if (op == UNARYMINUS) // Unary Op
                printf("%s = - %s\n",result, arg1);
            else // Copy
                printf("%s = %s\n",result, arg1);
    }
```

```
%{
#include <math.h>
#include "y.tab.h"
#include "parser.h"
%}

ID          [A-Za-z][A-Za-z0-9]*

%%
[0-9]+      {
              yylval.intval = atoi(yytext);
              return NUMBER;
            }

[ \t]       ;  /* ignore white space */


{ID}        { /* return symbol pointer */
              yylval.symp = symlook(yytext);
              return NAME;
            }

"$"         { return 0; /* end of input */ }

\n|.        return yytext[0];
%%
```

Sample Run

```
$ ./a.out
a = 2 + 3 * 4
b = (a + 5) / 6
c = (a + b) * (a - b) * -1
    t00 = 2
    t01 = 3
    t02 = 4
    t03 = t01 * t02
    t04 = t00 + t03
    a = t04
    t05 = 5
    t06 = a + t05
    t07 = 6
    t08 = t06 / t07
    b = t08
    t09 = a + b
    t10 = a - b
    t11 = t09 * t10
    t12 = 1
    t13 = - t12
    t14 = t11 * t13
    c = t14
$
```

Boolean Expression Grammar

1: $B \rightarrow B_1 \mid\mid B_2$
2: $B \rightarrow B_1 \;\&\&\; B_2$
3: $B \rightarrow !B_1$
4: $B \rightarrow (B_1)$
5: $B \rightarrow E_1 \texttt{ relop } E_2$
6: $B \rightarrow \texttt{true}$
7: $B \rightarrow \texttt{false}$

Attributes for Boolean expression

- *B.truelist*
  - List of (indices of) quads having dangling true exits or the Boolean expression
- *B.falselist*
  - List of (indices of) quads having dangling false exits or the Boolean expression
- *B.loc*
  - Location to store the value of the Boolean expression (optional)
- *nextinstr*
  - Global counter to the array of quads – the index of the next quad to be generated.

Boolean expression

- Control-flow translation of Boolean expressions

Production	Semantic Rules
B→B1 || B2	B1.true=B.true B1.false=newlabel() B2.true=B.true B2.false=B.false B.code=B1.code||label(B1.false)||B2.code
B→B1 && B2	B1.true=newlabel() B1.false=B.false B2.true=B.true B2.false=B.false B.code=B1.code||label(B1.true)||B2.code
B→!B1	B1.true=B.false B1.false=B.true B.code=B1.code
B→E1 rel E2	B.code=E1.code||E2.code||gen('if' E1.addr rel.op E2.addr 'goto' B.true)||gen('goto' B.false)
B→true	B.code=gen('goto' B.true)
B→false	B.code=gen('goto' B.false)

Control Flow

- Boolean expressions

  B → B || B | B && B | !B | (B) | E rel E | true | false

- Example

  if(x<100||x>200&&x!=y) x=0;

Avoid redundant goto statements

```
    if x<100 goto L2
    ifFalse x>200 goto L1
    ifFalse x!=y goto L1
L2: x=0
L1:
```

```
    if x<100 goto L2
    goto L3
L3: if x>200 goto L4
    goto L1
L4: if x!=y goto L2
    goto L1
L2: x=0
L1:
```

Boolean value and Jumping code

1. Use two passes:

   Construct the complete syntax tree as input, walk the tree in depth-first order, compute the translation following semantic rules.

2. Use one pass for statement, but two passes for expressions:

   Expression (construct syntax tree and walk the tree) should be translated before statement.

Boolean expression grammar with back-patching

1:	B	$\rightarrow$	$B_1 \mid\mid M\ B_2$
2:	B	$\rightarrow$	B_1 && $M\ B_2$
3:	B	$\rightarrow$	$!B_1$
4:	B	$\rightarrow$	(B_1)
5:	B	$\rightarrow$	E_1 `relop` E_2
6:	B	$\rightarrow$	`true`
7:	B	$\rightarrow$	`false`
8:	M	$\rightarrow$	ϵ

Methods for Back-patching

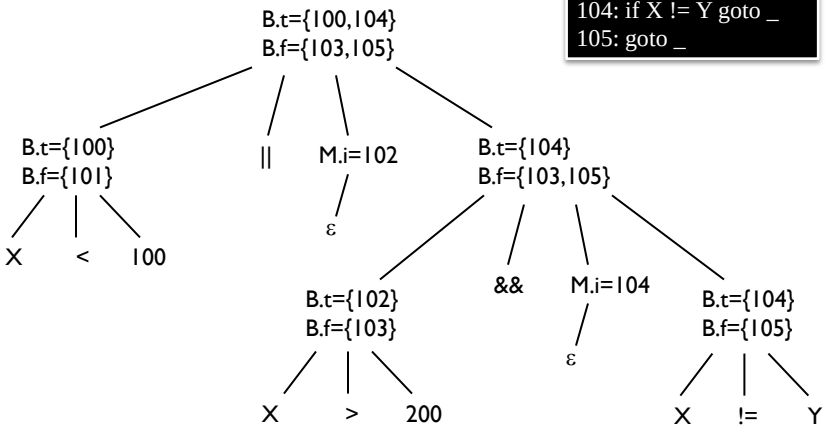
- *M.instr:* index of the quad generated at M.
- *makelist(i)*: creates a new list containing an index (*i*) into the array of instructions; *makelist* returns a pointer to the newly created list.
- *merge(p1,p2)*: concatenates *p1* list and *p2* list, and returns a pointer to the list.
- *backpatch(p,i)*: inserts *i* as the target label for each of the instructions on the list pointed to by *p*.

Back-patching with actions

B→B1 || M B2	{ backpatch(B1.falselist,M.instr); B.truelist=merge(B1.truelist,B2.truelist); B.falselist=B2.falselist; }
B→B1 && M B2	{ backpatch(B1.truelist,M.instr); B.truelist=B2.truelist; B.falselist=merge(B1.falselist,B2.falselist); }
B→!B1	{ B.truelist=B1.falselist; B.falselist=B1.truelist; }
B→(B1)	{ B.truelist=B1.truelist; B.falselist=B1.falselist; }
B→ E1 rel E2	{ B.truelist=makelist(nextinstr); B.falselist=makelist(nextinstr+1); gen('if E1.addr rel.op E2.addr 'goto _'); gen('goto _');}
B→true	{ B.truelist=makelist(nextinstr); gen('goto _'); }
B→false	{ B.falselist=makelist(nextinstr); gen('goto _'); }
M→ε	{ M.instr=nextinstr; }

Back-patching

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



X<100 || X>200 && X!=Y

Control Construct

- Grammar
  1. *S* → { *L* }
  2. *S* → *id* = *E* ;
  3. *S* → if (*B*) *S*
  4. *S* → if (*B*) *S* else *S*
  5. *S* → while (*B*) *S*
  6. *L* → *L S*
  7. *L* → *S*
- Attributes
  - *S.nextlist*: List of (indices of) quads having dangling exits for statements *S*.
  - *L.nextlist*: List of (indices of) quads having dangling exits for (list of) statements *L*.

Control Construct with back-patching

1: $S \rightarrow \{ L \}$
2: $S \rightarrow \textbf{id} = E ;$
3: $S \rightarrow \textbf{if} (B) M S_1$
4: $S \rightarrow \textbf{if} (B) M_1 S_1 N \textbf{ else } M_2 S_2$
5: $S \rightarrow \textbf{while } M_1 (B) M_2 S_1$
6: $L \rightarrow L_1 M S$
7: $L \rightarrow S$
8: $M \rightarrow \epsilon$
9: $N \rightarrow \epsilon$

Backpatching

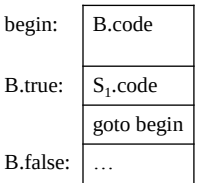
$S \rightarrow$ **if** (B) S | **if** (B) S **else** S | **while** (B) S | { L } | A ;
$L \rightarrow L\ S \mid S$

$S \rightarrow$ **if** (B) $M\ S_1$	{ backpatch(B.truelist,M.instr); S.nextlist=merge(B.falselist, S_1.nextlist); }
$S \rightarrow$ **if** (B) $M_1\ S_1\ N$ **else** $M_2\ S_2$	{ backpatch(B.truelist, M1.instr); backpatch(B.falselist, M_2.instr); temp=merge(S_1.nextlist,N.nextlist); S.nextlist=merge(temp,S_2.nextlist); }
$S \rightarrow$ **while** M_1 (B) $M_2\ S_1$	{ backpatch(S_1.nextlist, M_1.instr); backpatch(B.truelist, M_2.instr); S.nextlist=B.falselist; gen('goto' M_1.instr); }
$S \rightarrow \{L\}$	{ S.nextlist = L.nextlist; }
$S \rightarrow A$;	{ S.nextlist=null; }
$M \rightarrow \varepsilon$	{ M.instr=nextinstr; }
$N \rightarrow \varepsilon$	{ N.nextlist=makelist(nextinstr); gen('goto _'); }
$L \rightarrow L_1\ M\ S$	{ backpatch(L_1.nextlist,M.instr); L.nextlist=S.nextlist; }
$L \rightarrow S$	{ L.nextlist = S.nextlist; }

Control Flow

- Flow-of-control statements

  S → if (B) *S1* | if (B) *S1* else *S2* | while (B) *S1*



***if* and *if-else*?**

PRODUCTION	SEMANTIC RULES
$S \rightarrow \text{while}(B)S_1$	begin=newlabel() B.true=newlable() B.false=S.next S_1.next=begin S.code=label(begin)‖B.code‖ label(B.true)‖S_1.code‖ gen('goto' begin)

Control Flow

- Flow-of-control statements

  S → if (B) *S1* | if (B) *S1* else *S2* | while (B) *S1*

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	S.next=newlabel() P.code=S.code‖label(S.next)
$S \rightarrow$ assign	S.code=assign.code
$S \rightarrow \text{if } (B)\ S_1$	B.true=newlabel() B.false=S_1.next=S.next S.code=B.code‖label(B.true)‖S_1.code
$S \rightarrow \text{if } (B)\ S_1 \text{ else } S_2$	B.true=newlabel() B.false=newlabel() S_1.next=S_2.next=S.next S.code=B.code‖label(B.true)‖S_1.code‖gen('goto' S.next)‖ label(B.false)‖S_2.code
$S \rightarrow S_1\ S_2$	S_1.next=newlabel() S_2.next=S.next S.code=S_1.code‖label(S_1.next)‖S_2.code

Handling Goto

- Maintain a Label table having lookup(Label) and
  - ID of Label
    - Entered to Label table either when a label is defined or it used as a target for a goto before being used.
  - ADDR
    - Address of Label (index of quad))is set from the definition of a label. Hence it will be null as long as a label has been encountered in one or more goto's but not defined yet
  - LST
    - For this label the list of dangling goto will be null if ADDR is not null.

Handling Goto

```
L1:     /* if L1 exists in Label Table
            if(ADDR==NULL)
              ADDR=nextinstr
              backpatch LST with ADDR
              LST = null
            else
              duplicate definition of label L1 – an error
        if L1 does not exist, make an entry
          ADDR=nextinstr
          LST=null  */

goto L1;    /* If L1 exists in Lable Table
                 if (ADDR==null)   // Forward jump already noted
                   LST=merge(LST, makelist(nextinstr)
                 else   // Backward jump - target crossed
                   use ADDR
               If L1 does not exits, make an entry
                 ADDR=null  // New forward jump
                 LST = makelist(nextinstr)  */
```

Break, Continue and For

```
for ( ; ; readch() ) {
        if(peek==‘ ‘ || peek==‘\t’) continue;
        else if(peek==‘\n’) line = line +1;
        else break;
}
```

Back-patching – do it

$S \rightarrow$ do $M_1\ S_1\ M_2$ while (B);

$S \rightarrow$ for (E_1 ; $M_1\ B$; $M_2\ E_2\ N$) $M_3\ S_1$

Switch Statement

Example:

switch (E) {
case V_1: S_1
case V_2: S_2
case V_3: S_3
…..
default: S_n
}

Translation of Switch Statements

1. Evaluate the expression *E*
2. Find the value *Vj* in the list of cases that is the same as the value of he expression.
3. Execute the statement *Sj* associated with the value found.

Homework

- S → switch (*E*) S_1
- S → case num: S_1
- S → default: S_1