

Compiler Short Revision Notes

Sourabh Aggarwal

Compiled on April 20, 2019

Contents

- 1 Intro And ML Lex**
- 2 Parsing**
 - 2.1 LR(0)
 - 2.2 SLR
 - 2.3 LR(1)
- 3 CH 5 (Not in syllabus) Semantic Analysis**
- 4 CH 6 Activation Records**
- 5 CH 7 Translation To Intermediate Code**
- 6 CH 8 Basic Blocks And Traces**
- 7 CH 9 Instruction Selection**
- 8 CH 10 Liveness Analysis**
- 9 CH 11 Register Allocation**
- 10 CH 13 Garbage Collection**

1 Intro And ML Lex

some part is in class notes

2 $(a \odot b)|\epsilon$ represents the language {"", "ab"}. In writing regular expressions,
 3 we will sometimes omit the concatenation symbol or the epsilon,
 4 and we will assume that Kleene closure "binds tighter" than concatenation,
 5 and concatenation binds tighter than alternation; so that $ab|c$ means $(a \odot b)|c$, and $(a|)$ means $(a|\epsilon)$. Let us introduce some more abbreviations: [abed] means $(a|b|c|d)$, [b-g] means [bcdefg], [b-gM-Qkr] means [bcdefgMNOPQkr], M? means $(M|\epsilon)$, and M^+ means $(M \odot M^*)$.

```
5     if                               (IF);
6     [a-z][a-z0-9]*                  (ID);
7     [0-9]+                           (NUM);
8     ([0-9]+.“[0-9]*)|([0-9]*.“[0-9]+) (REAL);
9     ("--"[a-z]*`\n")|(" `"\n`"\t")+ (continue());
10    .                                (error(); continue());
```

FIGURE 2.2. Regular expressions for some tokens.

10 Longest match: The longest initial substring of the input that can match any regular expression is taken as the next token.

10 Rule priority: For a **particular** longest initial substring, the first regular expression that can match determines its token type. This means that the order of writing down the regular-expression rules has significance.

So according to the rules, `if8` match as a single identifier and not as the two tokens `if` and `8`. And `"if 89"` begin with a reserved word and not by an identifier by rule priority rule.

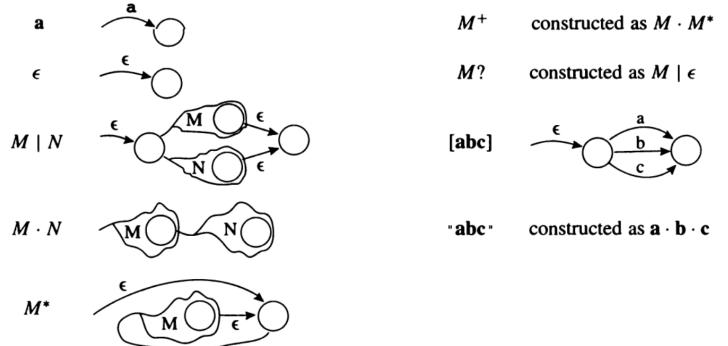


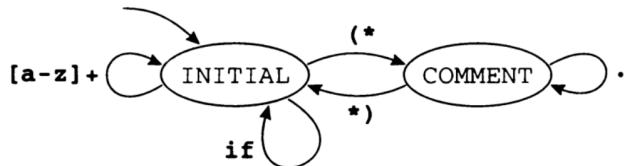
FIGURE 2.6. Translation of regular expressions to NFAs.

```

(* ML Declarations: *)
type lexresult = Tokens.token
fun eof() = Tokens.EOF(0,0)
%%
(* Lex Definitions: *)
digits=[0-9] +
%%
(* Regular Expressions and Actions: *)
if          => (Tokens.IF(yypos,yypos+2));
[a-z][a-z0-9]*  => (Tokens.ID(yytext,yypos,yypos+size yytext));
(digits)       => (Tokens.NUM(Int.fromString yytext,
                                yypos,yypos+size yytext));
(({digits}.."[0-9]*")|({0-9}*.{digits})
                      => (Tokens.REAL(Real.fromString yytext,
                                         yypos, yypos+size yytext));
(---[a-z]*"\n")|(" "!"\n"!"\t")+
                      => (continue());
                      => (ErrorMsg.error yypos "illegal character";
                           continue());

```

PROGRAM 2.9. ML-Lex specification of the tokens from Figure 2.2.



The ML-Lex specification corresponding to this machine is

the usual preamble ...

```
%%
% COMMENT
%%
<INITIAL>if      => (Tokens.IF(yypos,yypos+2));
<INITIAL>[a-z]+   => (Tokens.ID(yytext,yypos,
                                     yypos+size(yytext)));
<INITIAL>"(*"    => (YYBEGIN COMMENT; continue());
<COMMENT>"*)"   => (YYBEGIN INITIAL; continue());
<COMMENT>.       => (continue());
```

Certain rules

- An individual character stands for itself, except for the reserved characters
 $? * + | () ^ \$ / ; . = < > [{ " \ \$$
- A backslash followed by one of the reserved characters stands for that character.
- Inside the brackets, only the symbols

$\backslash - ^$

are reserved. An initial up-arrow \wedge stands for the complement of the characters listed, e.g. $[\wedge abc]$ stands any character except a, b, or c.

- To include \wedge literally in a bracketed set, put it anywhere but first; to include $-$ literally in a set, put it first or last.
- The dot $.$ character stands for any character except newline, i.e. the same as

$[\wedge \backslash n]$

- The following special escape sequences are available, inside or outside of square brackets:

```
\b backspace
\n newline
\t horizontal tab
\ddd where ddd is a 3 digit decimal escape
```

- Any regular expression may be enclosed in parentheses $()$ for syntactic (but, as usual, not semantic) effect
- A sequence of characters will stand for itself (reserved characters will be taken literally) if it is enclosed in double quotes $" "$.
- A postfix repetition range $\{a, b\}$ where a and b are small integers stands for any number of repetitions between a and b of the preceding expression. The notation $\{a\}$ stands for exactly a repetitions. Ex: $[0-9]\{3\}$ Any three-digit decimal number.
- The rules should match all possible input. If some input occurs that does not match any rule, the lexer created by ML-Lex will raise an exception LexError.
- The user may recursively call the lexing function with `lex()`. (If `%arg` is used, the lexing function may be re-invoked with the same argument by using `continue()`.) This is convenient for ignoring white space or comments silently:

```
[\ \t\n]+      => ( lex());
```

- To switch start states, the user may call YYBEGIN with the name of a start state.

2 Parsing

some part is in class notes

The parser returns an abstract syntax tree of the expression being evaluated. The parser gets tokens from the scanner to parse the input and build the AST. When an AST is returned by the parser, the compiler calls the code generator to evaluate the tree and produce the target code. There are two main parts to a compiler, the front end and back end. The front end reads the tokens and builds an AST of a program. The back end generates the code given the AST representation of the program. As presented in earlier chapters, the front end consists of the scanner and the parser.

As before, we say that a language is a set of strings; each string is a finite sequence of symbols taken from a finite alphabet. For parsing, the strings are source programs, the symbols are lexical tokens, and the alphabet is the set of token types returned by the lexical analyzer.

A leftmost derivation is one in which the leftmost nonterminal symbol is always the one expanded; in a rightmost derivation, the rightmost non-terminal is always next to be expanded.

A parse tree is made by connecting each symbol in a derivation to the one from which it was derived, as shown in Figure 3.3. Two different derivations can have the same parse tree.

A grammar is ambiguous if it can derive a sentence with two different parse trees.

Parsers must read not only terminal symbols such as $+$, $-$, num , and so on, but also the end-of-file marker. We will use $\$$ to represent end of file.

Suppose S is the start symbol of a grammar. To indicate that $\$$ must come after a complete S -phrase, we augment the grammar with a new start symbol S' and a new production $S' \rightarrow S\$$.

Predictive Parsing: Some grammars are easy to parse using a simple algorithm known as recursive descent. Predictive parsing works only on grammars where the first terminal symbol of each subexpression provides enough information to choose which production to use.

$S \rightarrow E \$$

$E \rightarrow E + T$	$T \rightarrow T * F$	$F \rightarrow id$
$E \rightarrow E - T$	$T \rightarrow T / F$	$F \rightarrow num$
$E \rightarrow T$	$T \rightarrow F$	$F \rightarrow (E)$

GRAMMAR 3.10.

$S \rightarrow if E then S else S$
 $S \rightarrow begin S L$
 $S \rightarrow print E$

$L \rightarrow end$
 $L \rightarrow ; S L$
 $E \rightarrow num = num$

GRAMMAR 3.11.

```
datatype token = IF | THEN | ELSE | BEGIN | END | PRINT
                | SEMI | NUM | EQ

val tok = ref (getTok())
fun advance() = tok := getTok()
fun eat(t) = if (!tok=t) then advance() else error()

fun S() = case !tok
            of IF => (eat(IF); E(); eat(THEN); S();
                         eat(ELSE); S())
            | BEGIN => (eat(BEGIN); S(); L())
            | PRINT => (eat(PRINT); E())
and L() = case !tok
            of END => (eat(END))
            | SEMI => (eat(SEMI); S(); L())
and E() = (eat (NUM); eat(EQ); eat(NUM))
```

With suitable definitions of `error` and `getTok`, this program will parse `ry` nicely.

Emboldened by success with this simple method, let us try it with Grammar 3.10:

```
fun S() = (E(); eat(EOF))
and E() = case !tok
            of ? => (E(); eat(PLUS); T())
            | ? => (E(); eat(MINUS); T())
            | ? => (T())
and T() = case !tok
            of ? => (T(); eat(TIMES); F())
            | ? => (T(); eat(DIV); F())
            | ? => (F())
and F() = case !tok
            of ID => (eat(ID))
            | NUM => (eat(NUM))
            | LPAREN => (eat(LPAREN); E(); eat(RPAREN))
```

Given a string γ of terminal and nonterminal symbols, $FIRST(\gamma)$ is the set of all terminal symbols that can begin any string derived from γ . For example, let $\gamma = T * F$. Any string of terminal symbols derived from γ must start with `id`, `num`, or `(`. Thus,

$FIRST(T * F) = \{id, num, (\}$

If two different productions $X \rightarrow \gamma_1$ and $X \rightarrow \gamma_2$ have the same left-hand-side symbol (X) and their right-hand sides have overlapping $FIRST$ sets, then the grammar cannot be parsed using predictive parsing.

With respect to a particular grammar, given a string γ of terminals and nonterminals,

- nullable(X) is true if X can derive the empty string.
- FIRST(γ) is the set of terminals that can begin strings derived from γ .
- FOLLOW(X) is the set of terminals that can immediately follow X . That is,

47

CHAPTER THREE. PARSING

$t \in \text{FOLLOW}(X)$ if there is any derivation containing Xt . This can occur if the derivation contains $X Y Z t$ where Y and Z both derive ϵ .

A precise definition of FIRST, FOLLOW, and nullable is that they are the smallest sets for which these properties hold:

For each terminal symbol Z , $\text{FIRST}[Z] = \{Z\}$.

for each production $X \rightarrow Y_1 Y_2 \dots Y_k$

if $Y_1 \dots Y_k$ are all nullable (or if $k = 0$)

then nullable[X] = true

for each i from 1 to k , each j from $i + 1$ to k

if $Y_1 \dots Y_{i-1}$ are all nullable (or if $i = 1$)

then $\text{FIRST}[X] = \text{FIRST}[X] \cup \text{FIRST}[Y_i]$

if $Y_{i+1} \dots Y_k$ are all nullable (or if $i = k$)

then $\text{FOLLOW}[Y_i] = \text{FOLLOW}[Y_i] \cup \text{FOLLOW}[X]$

if $Y_{i+1} \dots Y_{j-1}$ are all nullable (or if $i + 1 = j$)

then $\text{FOLLOW}[Y_i] = \text{FOLLOW}[Y_i] \cup \text{FIRST}[Y_j]$

Error recovery in predictive parsing (LL (1)):

A syntax error occurs when the string of input tokens is not a sentence in the language. Error recovery is a way of finding some sentence similar to that string of tokens. This can proceed by deleting, replacing, or inserting tokens.

For example, error recovery for T could proceed by inserting a `num` token.

It's not necessary to adjust the actual input; it suffices to pretend that the `num` was there, print a message, and return normally.

```
and T() = case !tok
  of ID      => (F(); T'())
  | NUM     => (F(); T'())
  | LPAREN => (F(); T'())
  | _       => print("expected id, num,
                    or left-paren")
```

It's a bit dangerous to do error recovery by insertion, because if the error cascades to produce another error, the process might loop infinitely. Error recovery by deletion is safer, because the loop must eventually terminate when end-of-file is reached.

Simple recovery by deletion works by skipping tokens until a token in the FOLLOW set is reached. For example, error recovery for T' could work like this:

```
and T'() = case !tok
  of PLUS   => ()
  | TIMES  => (eat(TIMES); F(); T'())
  | RPAREN => ()
  | EOF    => ()
  | _      => (print "expected +, *, right-paren,
                  or end-of-file";
                skipto[PLUS, TIMES, RPAREN, EOF])
```

```
and skipto(stop) =
  if member(!tok, stop) then ()
  else (eat(!tok); skipto(stop))
```

2.1 LR(0)

Closure(I) =

repeat

for any item $A \rightarrow \alpha.X\beta$ in I
 for any production $X \rightarrow \gamma$
 $I \leftarrow I \cup \{X \rightarrow \cdot\gamma\}$

until I does not change.

return I

Goto(I, X) =

set J to the empty set
 for any item $A \rightarrow \alpha.X\beta$ in I
 add $A \rightarrow \alpha X \cdot \beta$ to J
return $\text{Closure}(J)$

Note: 'X' in Goto can be either a terminal or non terminal.
 Below Grammar is same as that written in notebook.

Shift (n): Advance input one token; push n on stack.

Reduced (k): Pop stack as many times as the number of symbols on the right-hand side of rule k. Let X be the left-hand-side symbol of rule k; In the state now on top of stack, look up X to get "goto n"; Push n on top of stack.

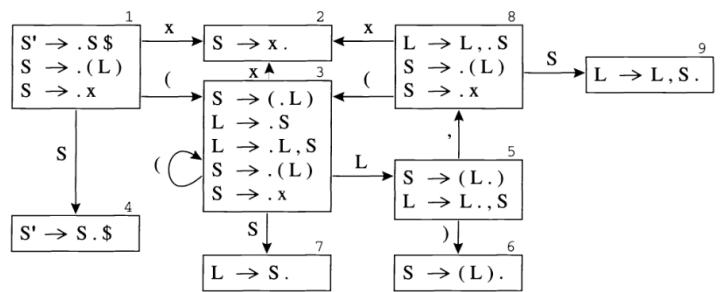


FIGURE 3.21. LR(0) states for Grammar 3.20.

	()	x	.	\$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

TABLE 3.22. LR(0) parsing table for Grammar 3.20.

Now here is the algorithm for LR(0) parser construction. First, augment the grammar with an auxiliary start production $S' \rightarrow S\$$. Let T be the set of states seen so far, and E the set of (shift or goto) edges found so far.

Initialize T to $\{\text{Closure}(\{S' \rightarrow S\$)\}\}$

Initialize E to empty.

repeat

for each state I in T

for each item $A \rightarrow \alpha.X\beta$ in I

let J be $\text{Goto}(I, X)$

$T \leftarrow T \cup \{J\}$

$E \leftarrow E \cup \{I \xrightarrow{X} J\}$

until E and T did not change in this iteration

However, for the symbol $\$$ we do not compute $\text{Goto}(I, \$)$; instead we will make an **accept** action.

For Grammar 3.20 this is illustrated in Figure 3.21.

Now we can compute set R of LR(0) reduce actions:

$R \leftarrow \{\}$

for each state I in T

for each item $A \rightarrow \alpha$ in I

$R \leftarrow R \cup \{(I, A \rightarrow \alpha)\}$

We can now construct a parsing table for this grammar (Table 3.22). For each edge $I \xrightarrow{X} J$ where X is a terminal, we put the action **shift** J at position (I, X) of the table; if X is a nonterminal we put **goto** J at position (I, X) . For each state I containing an item $S' \rightarrow S.\$$ we put an **accept** action at $(I, \$)$. Finally, for a state containing an item $A \rightarrow \cdot\gamma$ (production n with the dot at the end), we put a **reduce** n action at (I, Y) for every token Y .

2.2 SLR

```

 $R \leftarrow \{\}$ 
for each state  $I$  in  $T$ 
  for each item  $A \rightarrow \alpha$  in  $I$ 
    for each token  $X$  in FOLLOW( $A$ )
       $R \leftarrow R \cup \{(I, X, A \rightarrow \alpha)\}$ 
  
```

The action $(I, X, A \rightarrow \alpha)$ indicates that in state I , on lookahead symbol X , the parser will reduce by rule $A \rightarrow \alpha$.

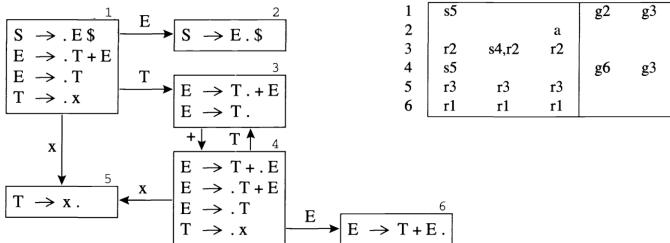
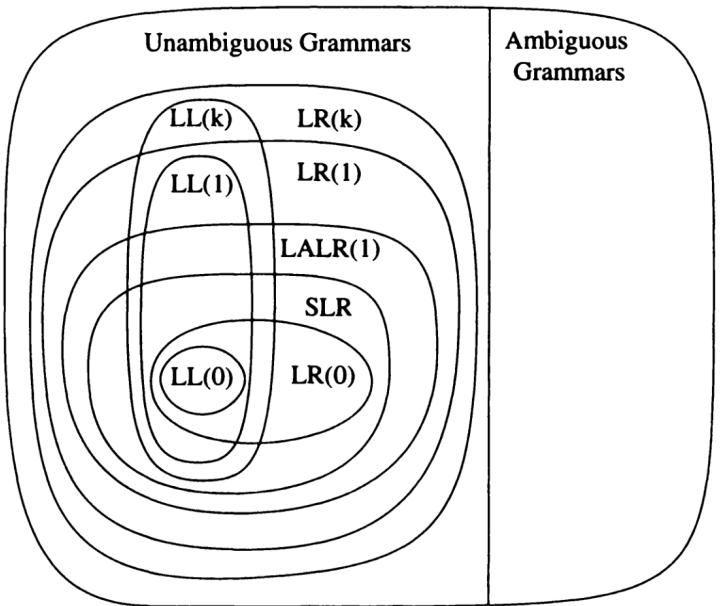


FIGURE 3.24. LR(0) states and parsing table for Grammar 3.23.

	x	+	\$	E	T
1	s5			g2	g3
2		a			
3	s4	r2			
4	s5			g6	g3
5	r3	r3	r3		
6	r1				

FIGURE 3.25. SLR parsing table for Grammar 3.23.



In examining a shift-reduce conflict such as

```

 $E \rightarrow E * E .$  +
 $E \rightarrow E . + E$  (any)
  
```

there is the choice of shifting a *token* and reducing by a *rule*. Should the rule or the token be given higher priority? The precedence declarations (%left, etc.) give priorities to the tokens; the priority of a rule is given by the last token occurring on the right-hand side of that rule. Thus the choice here is between a rule with priority * and a token with priority +; the rule has higher priority, so the conflict is resolved in favor of reducing.

When the rule and token have equal priority, then a %left precedence favors reducing, %right favors shifting, and %nonassoc yields an error action.

Instead of using the default “rule has precedence of its last token,” we can assign a specific precedence to a rule using the %prec directive. This is commonly used to solve the “unary minus” problem. In most programming languages a unary minus binds tighter than any binary operator, so $-6 * 8$ is parsed as $(-6) * 8$, not $- (6 * 8)$. Grammar 3.35 shows an example.

The token UMINUS is never returned by the lexer; it is merely a place-

2.3 LR(1)

Closure(I) =

```

repeat
  for any item  $(A \rightarrow \alpha.X\beta, z)$  in  $I$ 
    for any production  $X \rightarrow \gamma$ 
      for any  $w \in \text{FIRST}(\beta z)$ 
         $I \leftarrow I \cup \{(X \rightarrow \gamma, w)\}$ 
  until  $I$  does not change
return  $I$ 
  
```

Goto(I, X) =

```

 $J \leftarrow \{\}$ 
for any item  $(A \rightarrow \alpha.X\beta, z)$  in  $I$ 
  add  $(A \rightarrow \alpha.X.\beta, z)$  to  $J$ 
return Closure( $J$ ).
  
```

The start state is the closure of the item $(S' \rightarrow .S \$, ?)$, where the lookahead symbol ? will not matter, because the end-of-file marker will never be shifted.

The reduce actions are chosen by this algorithm:

```

 $R \leftarrow \{\}$ 
for each state  $I$  in  $T$ 
  for each item  $(A \rightarrow \alpha, z)$  in  $I$ 
     $R \leftarrow R \cup \{(I, z, A \rightarrow \alpha)\}$ 
  
```

```

%%
%term INT | PLUS | MINUS | TIMES | UMINUS | EOF
%nonterm exp
%start exp
%eof EOF

%left PLUS MINUS
%left TIMES
%left UMINUS
%%

exp : INT          ()
| exp PLUS exp   ()
| exp MINUS exp  ()
| exp TIMES exp  ()
| MINUS exp      %prec UMINUS ()
  
```

GRAMMAR 3.35.

holder in the chain of precedence (%left) declarations. The directive %prec UMINUS gives the rule exp: MINUS exp the highest precedence, so reducing by this rule takes precedence over shifting any operator, even a minus sign.

possible" semantic actions, especially if the actions contain side effects. Consider this grammar fragment:

```

statements: statements exp SEMICOLON
           | statements error SEMICOLON
           | (* empty *)

exp : increment exp decrement
     | ID

increment: LPAREN      (nest := nest + 1)
decrement: RPAREN      (nest := nest - 1)

```

"Obviously" it is true that whenever a semicolon is reached, the value of nest is zero, because it is incremented and decremented in a balanced way according to the grammar of expressions. But if a syntax error is found after some left parentheses have been parsed, then states will be popped from the stack without "completing" them, leading to a nonzero value of nest. The best solution to this problem is to have side-effect-free semantic actions that build abstract syntax trees, as described in Chapter 4.

Global error repair finds the smallest set of insertions and deletions that would turn the source string into a syntactically correct string, *even if the insertions and deletions are not at a point where an LL or LR parser would first report an error*. In this case, global error repair would do a single-token substitution, replacing type by var.

3 CH 5 (Not in syllabus) Semantic Analysis

This phase is characterized by the maintenance of symbol tables (also called environments) mapping identifiers to their types and locations.

An environment is a set of *bindings* denoted by the \mapsto arrow. For example, we could say that the environment σ_0 contains the bindings $\{g \mapsto \text{string}, a \mapsto \text{int}\}$; meaning that the identifier a is an integer variable and g is a string variable.

```

1   function f(a:int, b:int, c:int) =
2       (print_int(a+c);
3        let var j := a+b
4        var a := "hello"
5        in print(a); print_int(j)
6        end;
7        print_int(b)
8    )

```

Suppose we compile this program in the environment σ_0 . The formal parameter declarations on line 1 give us the table σ_1 equal to $\sigma_0 + \{a \mapsto \text{int}, b \mapsto \text{int}, c \mapsto \text{int}\}$, that is, σ_0 extended with new bindings for a, b, and c. The identifiers in line 2 can be looked up in σ_1 . At line 3, the table $\sigma_2 = \sigma_1 + \{j \mapsto \text{int}\}$ is created; and at line 4, $\sigma_3 = \sigma_2 + \{a \mapsto \text{string}\}$ is created.

How does the + operator for tables work when the two environments being "added" contain different bindings for the same symbol? When σ_2 and $\{a \mapsto \text{string}\}$ map a to int and string, respectively? To make the scoping rules work the way we expect them to in real programming languages, we want $\{a \mapsto \text{string}\}$ to take precedence. So we say that $X + Y$ for tables is not the same as $Y + X$; bindings in the right-hand table override those in the left.

Finally, in line 6 we discard σ_3 and go back to σ_1 for looking up the identifier b in line 7. And at line 8, we discard σ_1 and go back to σ_0 .

4 CH 6 Activation Records

For the remainder of this chapter we will consider languages with stackable local variables and postpone discussion of higher-order functions to Chapter 15.

In many languages (including C, Pascal, and Tiger), local variables are destroyed when a function returns. Since a function returns only after all the functions it has called have returned, we say that function calls behave in last-in-first-out (LIFO) fashion. If local variables are created on function entry and destroyed on function exit, then we can use a LIFO data structure – a stack – to hold them.

```

fun f(x) =
  let fun g(y) = x+y
  in g
  end

val h = f(3)
val j = f(4)

val z = h(5)
val w = j(7)

```

(a) Written in ML

```

int (*)() f(int x) {
  int g(int y) {return x+y;}
  return g;
}

int (*h)() = f(3);
int (*j)() = f(4);

int z = h(5);
int w = j(7);

```

(b) Written in pseudo-C

PROGRAM 6.1. An example of higher-order functions.

But in languages supporting both nested functions *and* function-valued variables, it may be necessary to keep local variables after a function has returned!

It is the combination of *nested functions* (where inner functions may use variables defined in the outer functions) and *functions returned as results* (or stored into variables) that causes local variables to need lifetimes longer than their enclosing function invocations.

Pascal (and Tiger) have nested functions, but they do not have functions as returnable values. C has functions as returnable values, but not nested functions. So these languages can use stacks to hold local variables.

ML, Scheme, and several other languages have both nested functions and functions as returnable values (this combination is called *higher-order functions*). So they cannot use stacks to hold all local variables. This complicates

Instead, we treat the stack as a big array, with a special register – the *stack pointer* – that points at some location. All locations beyond the stack pointer are considered to be garbage, and all locations before the stack pointer are considered to be allocated. The stack usually grows only at the entry to a function, by an increment large enough to hold all the local variables for that function, and, just before the exit from the function, shrinks by the same amount. The area on the stack devoted to the local variables, parameters, return address, and other temporaries for a function is called the function's *activation record* or *stack frame*. For historical reasons, run-time stacks usually start at a high memory address and grow toward smaller addresses. This can be rather confusing: stacks grow downward and shrink upward, like icicles.

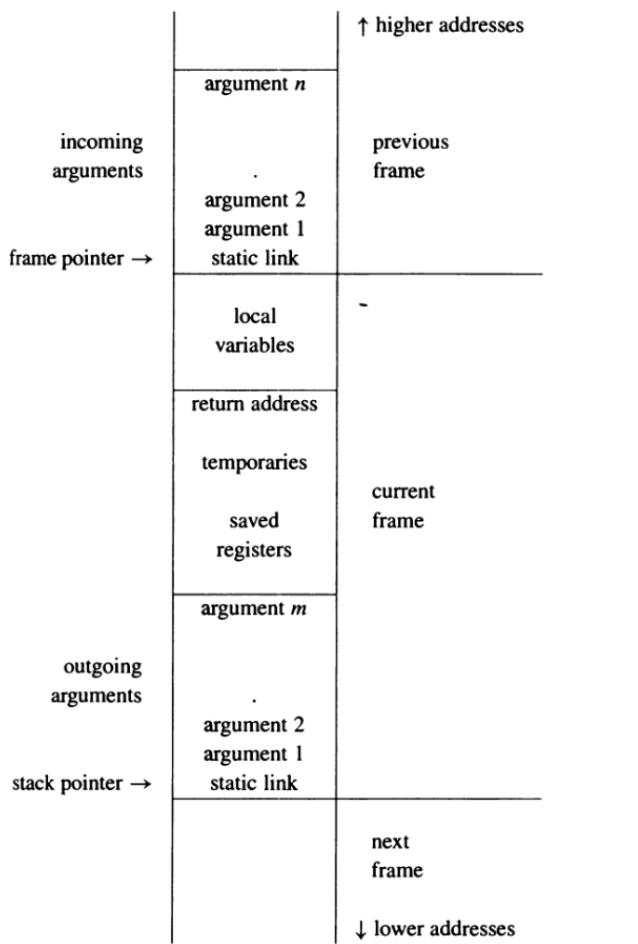


FIGURE 6.2. A stack frame.

Figure 6.2 shows a typical stack frame layout. The frame has a set of incoming arguments (technically these are part of the previous frame but they are at a known offset from the frame pointer) passed by the caller. The return address is created by the CALL instruction and tells where (within the calling function) control should return upon completion of the current function. Some local variables are in this frame; other local variables are kept in machine registers. Sometimes a local variable kept in a register needs to be saved into the frame to make room for other uses of the register; there is an area in the frame for this purpose. Finally, when the current function calls other functions, it can use the outgoing argument space to pass parameters.

Suppose a function $g(\dots)$ calls the function $f(a_1, \dots, a_n)$. We say g is the caller and f is the callee. On entry to f , the stack pointer points to the first argument that g passes to f . On entry, f allocates a frame by simply subtracting the frame size from the stack pointer SP. The old SP becomes the current frame pointer FP.

FP is a "fictional" register whose value is always SP+framesize. Why talk about a frame pointer at all? Why not just refer to all variables, parameters, etc. by their offset from SP, if the frame size is constant? The frame size is not known until quite late in the compilation process, when the number of memory-resident temporaries and saved registers is determined.

Suppose a function f is using register r to hold a local variable and calls procedure g , which also uses r for its own calculations. Then r must be saved (stored into a stack frame) before g uses it and restored (fetched back from the frame) after g is finished using it. But is it f 's responsibility to save and restore the register, or g 's? We say that r is a caller-save register if the caller (in this case, f) must save and restore the register, and r is callee-save if it is the responsibility of the callee (in this case, g). therefore the first k arguments (for $k = 4$ or $k = 6$, typically) of a function are passed in registers r_p, \dots, r_{p+k-1} and the rest of the arguments are passed in memory.

Now, suppose $f(a_1, \dots, a_n)$ (which received its parameters in r_1, \dots, r_n , for example) calls $h(z)$. It must pass the argument z in r_1 ; so f saves the old contents of r_1 (the value a_1) in its stack frame before calling h . But there is the memory traffic that was supposedly avoided by passing arguments in registers! How has the use of registers saved any time?

There are four answers, any or all of which can be used at the same time:

1. Some procedures don't call other procedures – these are called *leaf* procedures. What proportion of procedures are leaves? Well, if we make the (optimistic) assumption that the average procedure calls either no other procedures or calls at least two others, then we can describe a "tree" of procedure calls in which there are more leaves than internal nodes. This means that *most* procedures called are leaf procedures.
2. Leaf procedures need not write their incoming arguments to memory. In fact, often they don't need to allocate a stack frame at all. This is an important savings.
3. Some optimizing compilers use *interprocedural register allocation*, analyzing all the functions in an entire program at once. Then they assign different procedures different registers in which to receive parameters and hold local variables. Thus $f(x)$ might receive x in r_1 , but call $h(z)$ with z in r_7 .
4. Even if f is not a leaf procedure, it might be finished with all its use of the argument x by the time it calls h (technically, x is a dead variable at the point where h is called). Then f can overwrite r_1 without saving it.

If f needs to write an incoming parameter into the frame, where in the frame should it go? Ideally, f 's frame layout should matter only in the implementation of f . A straightforward approach would be for the caller to pass arguments a_1, \dots, a_k in registers and a_{k+1}, \dots, a_n at the end of its own frame – the place marked *outgoing arguments* in Figure 6.2 – which become the *incoming arguments* of the callee. If the callee needed to write any of these arguments to memory, it would write them to the area marked *local variables*.

To resolve the contradiction that parameters are passed in registers, but have addresses too, the first k parameters are passed in registers; but any parameter whose address is taken must be written to a memory location on entry to the function. To satisfy printf, the memory locations into which register arguments are written must all be consecutive with the memory locations in which arguments $k+1, k+2$, etc. are written. Therefore, C programs can't have some of the arguments saved in one place and some saved in another – they must all be saved contiguously.

When function g calls function f , eventually f must return. It needs to know where to go back to. If the call instruction within g is at address a , then (usually) the right place to return to is $a+1$, the next instruction in g . This is called the return address.

On modern machines, the call instruction merely puts the return address (the address of the instruction after the call) in a designated register. A non-leaf procedure will then have to write it to the stack (unless interprocedural register allocation is used), but a leaf procedure will not.

Many of the local variables will be allocated to registers, as will the intermediate results of expression evaluation. Values are written to memory (in the stack frame) only when necessary for one of these reasons:

- the variable will be passed by reference, so it must have a memory address (or, in the C language the & operator is anywhere applied to the variable);
- the variable is accessed by a procedure nested inside the current one;²
- the value is too big to fit into a single register;³
- the variable is an array, for which address arithmetic is necessary to extract components;
- the register holding the variable is needed for a specific purpose, such as parameter passing (as described above), though a compiler may move such values to other registers instead of storing them in memory;
- or there are so many local variables and temporary values that they won't all fit in registers, in which case some of them are "spilled" into the frame.

We will say that a variable escapes if it is passed by reference, its address is taken (using C's & operator), or it is accessed from a nested function. Unfortunately, the conditions in our list don't manifest themselves early enough. When the compiler first encounters the declaration of a variable, it doesn't yet know whether the variable will ever be passed by reference, accessed in a nested procedure, or have its address taken; and doesn't know how many registers the calculation of expressions will require. An industrial-strength compiler must assign provisional locations to all formals and locals, and decide later which of them should really go in registers.

In languages that allow nested function declarations, the inner functions may use variables declared in outer functions. This language feature is called block structure.

There are several methods to allow inner functions to access outer ones:

- Whenever a function f is called, it can be passed a pointer to the frame of the function statically enclosing f ; this pointer is the *static link*.
- A global array can be maintained, containing – in position i – a pointer to the frame of the most recently entered procedure whose *static nesting depth* is i . This array is called a *display*.
- When g calls f , each variable of g that is actually accessed by f (or by any function nested inside f) is passed to f as an extra argument. This is called *lambda lifting*.

I will describe in detail only the method of static links.

```

1      type tree = {key: string, left: tree, right: tree}
2
3      function prettyprint(tree: tree) : string =
4          let
5              var output := ""
6
7              function write(s: string) =
8                  output := concat(output, s)
9
10             function show(n:int, t: tree) =
11                 let function indent(s: string) =
12                     (for i := 1 to n
13                         do write(" "));
14                     output := concat(output, s); write("\n"))
15                 in if t=nil then indent(..)
16                 else (indent(t.key);
17                         show(n+1,t.left);
18                         show(n+1,t.right))
19             end
20
21             in show(0,tree); output
22         end

```

PROGRAM 6.3. Nested functions.

Line #

21 prettyprint calls show, passing prettyprint's own frame pointer as show's static link.

133

CHAPTER SIX. ACTIVATION RECORDS

- 10 show stores its static link (the address of prettyprint's frame) into its own frame.
 15 show calls indent, passing its own frame pointer as indent's static link.
 17 show calls show, passing its own static link (not its own frame pointer) as the static link.
 12 indent uses the value n from show's frame. To do so, it fetches at an appropriate offset from indent's static link (which points at the frame of show).
 13 indent calls write. It must pass the frame pointer of prettyprint as the static link. To obtain this, it first fetches at an offset from its own static link (from show's frame), the static link that had been passed to show.
 14 indent uses the variable output from prettyprint's frame. To do so it starts with its own static link, then fetches show's, then fetches output.⁴

```

signature FRAME =
sig type frame
    type access
    val newFrame : {name: Temp.label,
                   formals: bool list} -> frame
    val name : frame -> Temp.label
    val formals : frame -> access list
    val allocLocal : frame -> bool -> access
    :
end

```

5 CH 7 Translation To Intermediate Code

The front end of the compiler does lexical analysis, parsing, semantic analysis, and translation to intermediate representation. The back end does optimization of the intermediate representation and translation to machine language.

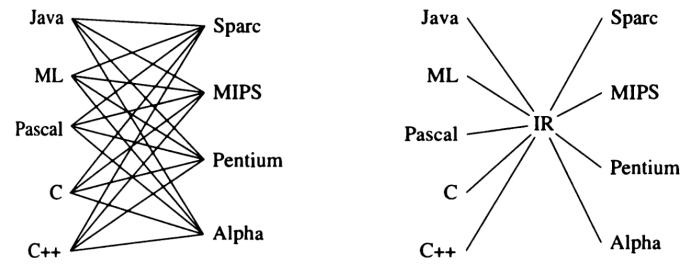


FIGURE 7.1. Compilers for five languages and four target machines: (left) without an IR, (right) with an IR.

For our compiler the intermediate representation tree language is defined by the signature tree, as shown in figure

```

signature TREE =
sig

datatype exp = CONST of int
| NAME of Temp.label
| TEMP of Temp.temp
| BINOP of binop * exp * exp
| MEM of exp
| CALL of exp * exp list
| ESEQ of stm * exp

and stm = MOVE of exp * exp
| EXP of exp
| JUMP of exp * Temp.label list
| CJUMP of relop * exp * exp * Temp.label * Temp.label
| SEQ of stm * stm
| LABEL of Temp.label

and binop = PLUS | MINUS | MUL | DIV
| AND | OR | LSHIFT | RSHIFT | ARSHIFT | XOR
| ULT | ULE | UGT | UGE
end

```

FIGURE 7.2. Intermediate representation trees.

Meaning:

expressions (exp), which stand for the computation of some value (possibly with side effects):

- CONST(i) The integer constant i .
 NAME(n) The symbolic constant n (corresponding to an assembly language label).
 TEMP(t) Temporary t . A temporary in the abstract machine is similar to a register in a real machine. However, the abstract machine has an infinite number of temporaries.
 BINOP(o, e_1, e_2) The application of binary operator o to operands e_1, e_2 . Subexpression e_1 is evaluated before e_2 . The integer arithmetic operators are PLUS, MINUS, MUL, DIV; the integer bitwise logical operators are AND, OR, XOR; the integer logical shift operators are LSHIFT, RSHIFT; the integer arithmetic right-shift is ARSHIFT. The Tiger language has no logical operators, but the intermediate language is meant to be independent of any source language; also, the logical operators might be used in implementing other features of Tiger.
 MEM(e) The contents of $wordSize$ bytes of memory starting at address e (where

$wordSize$ is defined in the Frame module). Note that when MEM is used as the left child of a MOVE, it means "store," but anywhere else it means "fetch."
 CALL(f, l) A procedure call: the application of function f to argument list l . The subexpression f is evaluated before the arguments which are evaluated left to right.

ESEQ(s, e) The statement s is evaluated for side effects, then e is evaluated for a result.

The statements (stm) of the tree language perform side effects and control flow:

MOVE(TEMP *t*, *e*) Evaluate *e* and move it into temporary *t*.

MOVE(MEM(*e*₁), *e*₂) Evaluate *e*₁, yielding address *a*. Then evaluate *e*₂, and store the result into *wordSize* bytes of memory starting at *a*.

EXP(*e*) Evaluate *e* and discard the result.

JUMP(*e*, *labs*) Transfer control (jump) to address *e*. The destination *e* may be a literal label, as in NAME(*lab*), or it may be an address calculated by any other kind of expression. For example, a C-language switch(*i*) statement may be implemented by doing arithmetic on *i*. The list of labels *labs* specifies all the possible locations that the expression *e* can evaluate to; this is necessary for dataflow analysis later. The common case of jumping to a known label *l* is written as JUMP(NAME *l*, [*l*]).

CJUMP(*o*, *e*₁, *e*₂, *t*, *f*) Evaluate *e*₁, *e*₂ in that order, yielding values *a*, *b*. Then compare *a*, *b* using the relational operator *o*. If the result is true, jump to *t*; otherwise jump to *f*. The relational operators are EQ and NE for integer equality and nonequality (signed or unsigned); signed integer inequalities LT, GT, LE, GE; and unsigned integer inequalities ULT, ULE, UGT, UGE.

SEQ(*s*₁, *s*₂) The statement *s*₁ followed by *s*₂.

LABEL(*n*) Define the constant value of name *n* to be the current machine code address. This is like a label definition in assembly language. The value NAME(*n*) may be the target of jumps, calls, etc.

What should the representation of an abstract syntax expression Ab-syn.exp be in the Tree language? At first it seems obvious that it should be Tree.exp. However, this is true only for certain kinds of expressions, the ones that compute a value. Expressions that return no value (such as some procedure calls, or while expressions in the Tiger language) are more naturally represented by Tree.stm. And expressions with Boolean values, such as *a* > *b*, might best be represented as a conditional jump - a combination of Tree.stm and a pair of destinations represented by Temp.labels. Therefore, we will make a datatype exp in the Translate module to model these three kinds of expressions:

```
datatype exp = Ex of Tree.exp
             | Nx of Tree.stm
             | Cx of Temp.label * Temp.label -> Tree.stm
```

Ex stands for an “expression,” represented as a Tree.exp.

Nx stands for “no result,” represented as a Tree.stm.

Cx stands for “conditional,” represented as a function from label-pair to statement. If you pass it a true-destination and a false-destination, it will make a statement that evaluates some conditionals and then jumps to one of the destinations (the statement will never “fall through”).

For example, the Tiger expression *a* > *b* | *c* < *d* might translate to the conditional:

```
Cx(fn (t, f) => SEQ(CJUMP(GT, a, b, t, z),
                      SEQ(LABEL z, CJUMP(LT, c, d, t, f))))
```

for some new label *z*.

Sometimes we will have an expression of one kind and we will need to convert it to an equivalent expression of another kind. For example, the Tiger statement

flag := (*a* > *b* | *c* < *d*)

```
structure T = Tree

fun unEx (Ex e) = e
| unEx (Cx genstm) =
  let val r = Temp.newtemp()
      val t = Temp.newlabel() and f = Temp.newlabel()
      in T.ESEQ(seq[T.MOVE(T.TEMP r, T.CONST 1),
                    genstm(t,f),
                    T.LABEL f,
                    T.MOVE(T.TEMP r, T.CONST 0),
                    T.LABEL t],
                T.TEMP r)
  end
| unEx (Nx s) = T.ESEQ(s, T.CONST 0)
```

FIGURE 7.3. The conversion function unEx.

requires the conversion of a Cx into an Ex so that a 1 (for true) or 0 (for false) can be stored into flag.

It is helpful to have three conversion functions:

```
unEx : exp -> Tree.exp
unNx : exp -> Tree.stm
unCx : exp -> (Temp.label * Temp.label -> Tree.stm)
```

Suppose *e* is the representation of *a* > *b* | *c* < *d*, so

e = Cx(fn(t, f) => ...)

Then the assignment statement can be implemented as

MOVE(TEMP_{flag}, unEx(*e*)).

The functions unCx and unNx are left as an exercise. It’s helpful to have unCx treat the cases of CONST 0 and CONST 1 specially, since they have particularly simple and efficient translations. Also, unCx(Nx _) need not be translated, as it should never occur in compiling a well typed Tiger program.

6 CH 8 Basic Blocks And Traces

It’s useful to be able to evaluate the subexpressions of an expression in any order. If tree expressions did not contain ESEQ and CALL nodes, then the order of evaluation would not matter.

Some of the mismatches between Trees and machine-language programs are:

- The CJUMP instruction can jump to either of two labels, but real machines’ conditional jump instructions fall through to the next instruction if the condition is false.
- ESEQ nodes within expressions are inconvenient, because they make different orders of evaluating subtrees yield different results.
- CALL nodes within expressions cause the same problem.
- CALL nodes within the argument-expressions of other CALL nodes will cause problems when trying to put arguments into a fixed set of formal-parameter registers.

We can take any tree and rewrite it into an equivalent tree without any of the cases listed above. Without these cases, the only possible parent of a SEQ node is another SEQ; all the SEQ nodes will be clustered at the top of the tree. This makes the SEQs entirely uninteresting; we might as well get rid of them and make a linear list of Tree.stms.

The transformation is done in three stages: First, a tree is rewritten into a list of canonical trees without SEQ or ESEQ nodes; then this list is grouped into a set of basic blocks, which contain no internal jumps or labels; then the basic blocks are ordered into a set of traces in which every CJUMP is immediately followed by its false label.

Thus the module Canon has these tree-rearrangement functions:

```
signature CANON =
sig
  val linearize : Tree.stm -> Tree.stm list
  val basicBlocks : Tree.stm list ->
    (Tree.stm list list * Temp.label)
  val traceSchedule : Tree.stm list list * Temp.label ->
    Tree.stm list
end

structure Canon : Canon
```

Linearize removes the ESEQs and moves the CALLS to top level. Then BasicBlocks groups statements into sequences of straight-line code. Finally traceSchedule orders the blocks so that every CJUMP is followed by its false label.

How can the ESEQ nodes be eliminated? The idea is to lift them higher and higher in the tree, until they can become SEQ nodes.

CANONICAL TREES

Let us define *canonical trees* as having these properties:

1. No SEQ or ESEQ.
2. The parent of each CALL is either EXP(...) or MOVE(TEMP *t*, ...).

(1)		$ESEQ(s_1, ESEQ(s_2, e)) = ESEQ(SEQ(s_1, s_2), e)$
(2)		$BINOP(op, ESEQ(s, e_1), e_2) = ESEQ(s, BINOP(op, e_1, e_2))$
	$MEM(ESEQ(s, e_1)) = ESEQ(s, MEM(e_1))$	
	$JUMP(ESEQ(s, e_1)) = SEQ(s, JUMP(e_1))$	
	$CJUMP(op, ESEQ(s, e_1), e_2, l_1, l_2) = SEQ(s, CJUMP(op, e_1, e_2, l_1, l_2))$	
(3)		$BINOP(op, e_1, ESEQ(s, e_2)) = ESEQ(MOVE(TEMP t, e_1), ESEQ(s, BINOP(op, TEMP t, e_2)))$
	$CJUMP(op, e_1, ESEQ(s, e_2), l_1, l_2) = SEQ(MOVE(TEMP t, e_1), SEQ(s, CJUMP(op, TEMP t, e_2, l_1, l_2)))$	
(4)		$BINOP(op, e_1, ESEQ(s, e_2)) = ESEQ(s, BINOP(op, e_1, e_2))$
	$CJUMP(op, e_1, ESEQ(s, e_2), l_1, l_2) = SEQ(s, CJUMP(op, e_1, e_2, l_1, l_2))$	

It may happen that *s* causes no side effects that can alter the result produced by *e*₁. This will happen if the temporaries and memory locations assigned by *s* are not referenced by *e*₁ (and *s* and *e*₁ don't both perform external I/O). In this case, identity (4) can be used.

We cannot always tell if two expressions commute. For example, whether MOVE(MEM(*x*), *y*) commutes with MEM(*z*) depends on whether *x* = *z*, which we cannot always determine at compile time. So we *conservatively approximate* whether statements commute, saying either “they definitely do commute” or “perhaps they don't commute.” For example, we know that any statement “definitely commutes” with the expression CONST(*n*), so we can use identity (4) to justify special cases like

$$BINOP(op, CONST(n), ESEQ(s, e)) = ESEQ(s, BINOP(op, CONST(n), e)).$$

The commute function estimates (very naively) whether two expressions commute:

```
fun commute(T.EXP(T.CONST _), _) = true
| commute(_, T.NAME _) = true
| commute(_, T.CONST _) = true
| commute _ = false
```

The reorder function takes a list of expressions and returns a pair of (statement, expression-list). The statement contains all the things that must be executed before the expression-list. As shown in these examples, this includes all the statement-parts of the ESEQs, as well as any expressions to their left with which they did not commute. When there are no ESEQs at all we will use EXP(CONST 0), which does nothing, as the statement.

Algorithm. Step one is to make a “subexpression-extraction” method for each kind. Step two is to make a “subexpression-insertion” method: given an ESEQ-clean version of each subexpression, this builds a new version of the expression or statement.

ML makes this easy to express using little “fn” functions:

```
val reorder_stm: Tree.exp list * (Tree.exp list -> Tree.stm)
  -> Tree.stm

val reorder_exp: Tree.exp list * (Tree.exp list -> Tree.exp)
  -> Tree.stm * Tree.exp

fun do_stm(T.JUMP(e, labs)) =
  reorder_stm([e], fn [e] => T.JUMP(e, labs))
| do_stm(T.CJUMP(p, a, b, t, f)) =
  reorder_stm([a, b], fn [a, b] => T.CJUMP(p, a, b, t, f))
| _ and so on

and do_exp(T.BINOP(p, a, b)) =
  reorder_exp([a, b], fn [a, b] => T.BINOP(p, a, b))
| do_exp(T.MEM(a)) =
  reorder_exp([a], fn [a] => T.MEM(a))
| _ and so on
```

Reorder_stm takes two arguments – a list *l* of subexpressions and a *build* function. It pulls all the ESEQs out of the *l*, yielding a statement *s*₁ that con-

tains all the statements from the ESEQs and a list *l*' of cleaned-up expressions. Then it makes SEQ(*s*₁, *build*(*l*')).

Reorder_exp(*l*, *build*) is similar, except that it returns a pair (*s*, *e*) where *s* is a statement containing all the side effects pulled out of *l*, and *e* is *build*(*l*')).

The left-hand operand of the MOVE statement is not considered a subexpression, because it is the *destination* of the statement – its value is not used by the statement. However, if the destination is a memory location, then the *address* acts like a source. Thus we have,

```
| do_stm(T.MOVE(T.TEMP t, b)) =
  reorder_stm([b], fn [b] => T.MOVE(T.TEMP t, b))
| do_stm(T.MOVE(T.MEM e, b)) =
  reorder_stm([e, b], fn [e, b] => T.MOVE(T.MEM e, b))
```

Now, given a list of extracted subexpressions, we pull the ESEQs out, from right to left.

MOVING CALLS TO TOP LEVEL

The Tree language permits CALL nodes to be used as subexpressions. However, the actual implementation of CALL will be that each function returns its result in the same dedicated return-value register TEMP(RV). Thus, if we have

`BINOP(PLUS, CALL(...), CALL(...))`

the second call will overwrite the RV register before the PLUS can be executed.

We can solve this problem with a rewriting rule. The idea is to assign each return value immediately into a fresh temporary register, that is

`CALL(fun, args) → ESEQ(MOVE(TEMP t, CALL(fun, args)), TEMP t)`

The rewriting rule is implemented as follows: `reorder` replaces any occurrence of `CALL(f, args)` by

`ESEQ(MOVE(TEMP tnew, CALL(f, args)), TEMP tnew)`

and calls itself again on the ESEQ. But `do_stm` recognizes the pattern

`MOVE(TEMP tnew, CALL(f, args)),`

and does not call `reorder` on the CALL node in that case, but treats the `f` and `args` as the children of the MOVE node. Thus, `reorder` never "sees"

A LINEAR LIST OF STATEMENTS

Once an entire function body s_0 is processed with `do_stm`, the result is a tree s'_0 where all the SEQ nodes are near the top (never underneath any other kind of node). The `linearize` function repeatedly applies the rule

$\text{SEQ}(\text{SEQ}(a, b), c) = \text{SEQ}(a, \text{seq}(b, c))$

The result is that s'_0 is linearized into an expression of the form

$\text{SEQ}(s_1, \text{SEQ}(s_2, \dots, \text{SEQ}(s_{n-1}, s_n) \dots))$

Here the SEQ nodes provide no structuring information at all, and we can just consider this to be a simple list of statements,

$s_1, s_2, \dots, s_{n-1}, s_n$

where none of the s_i contain SEQ or ESEQ nodes.

These rewrite rules are implemented by `linearize`, with an auxiliary function `linear`:

```
fun linearize(stm0: T.stm) : T.stm list =
let
  : definitions of reorder_exp, reorder_stm, do_exp, do_stm, etc.

  fun linear(T.SEQ(a,b),l) = linear(a,linear(b,l))
  | linear(s,l) = s::l
  in
    linear(do_stm stm0, nil)
  end
```

A *basic block* is a sequence of statements that is always entered at the beginning and exited at the end, that is:

- The first statement is a LABEL.
- The last statement is a JUMP or CJUMP.
- There are no other LABELS, JUMPs, or CJUMPs.

The algorithm for dividing a long sequence of statements into basic blocks is quite simple. The sequence is scanned from beginning to end; whenever a LABEL is found, a new block is started (and the previous block is ended); whenever a JUMP or CJUMP is found, a block is ended (and the next block is started). If this leaves any block not ending with a JUMP or CJUMP, then a JUMP to the next block's label is appended to the block. If any block has been left without a LABEL at the beginning, a new label is invented and stuck there.

We will apply this algorithm to each function-body in turn. The procedure "epilogue" (which pops the stack and returns to the caller) will not be part of this body, but is intended to follow the last statement. When the flow of program execution reaches the end of the last block, the epilogue should follow. But it is inconvenient to have a "special" block that must

come last and that has no JUMP at the end. Thus, we will invent a new label done - intended to mean the beginning of the epilogue - and put a JUMP(NAME done) at the end of the last block.

Now the basic blocks can be arranged in any order, and the result of executing the program will be the same - every block ends with a jump to the appropriate place. We can take advantage of this to choose an ordering of the blocks satisfying the condition that each CJUMP is followed by its false label.

At the same time, we can also arrange that many of the unconditional jumps are immediately followed by their target label. This will allow the deletion of these jumps, which will make the compiled program run a bit faster.

A *trace* is a sequence of statements that could be consecutively executed during the execution of the program. It can include conditional branches. A program has many different, overlapping traces. For our purposes in arranging CJUMPs and false-labels, we want to make a set of traces that exactly covers the program: each block must be in exactly one trace. To minimize the number of JUMPs from one trace to another, we would like to have as few traces as possible in our covering set.

A very simple algorithm will suffice to find a covering set of traces. The idea is to start with some block – the beginning of a trace – and follow a possible execution path – the rest of the trace. Suppose block b_1 ends with a JUMP to b_4 , and b_4 has a JUMP to b_6 . Then we can make the trace b_1, b_4, b_6 .

But suppose b_6 ends with a conditional jump CJUMP($cond, b_7, b_3$). We cannot know at compile time whether b_7 or b_3 will be next. But we can assume that some execution will follow b_3 , so let us imagine it is that execution that we are simulating. Thus, we append b_3 to our trace and continue with the rest of the trace after b_3 . The block b_7 will be in some other trace.

Put all the blocks of the program into a list Q .

while Q is not empty

 Start a new (empty) trace, call it T .

 Remove the head element b from Q .

while b is not marked

 Mark b ; append b to the end of the current trace T .

 Examine the successors of b (the blocks to which b branches);
 if there is any unmarked successor c

$b \leftarrow c$

 End the current trace T .

ALGORITHM 8.2. Generation of traces.

FINISHING UP

An efficient compiler will keep the statements grouped into basic blocks, because many kinds of analysis and optimization algorithms run faster on (relatively few) basic blocks than on (relatively many) individual statements. For the Tiger compiler, however, we seek simplicity in the implementation of later phases. So we will flatten the ordered list of traces back into one long list of statements.

At this point, most (but not all) CJUMPs will be followed by their true or false label. We perform some minor adjustments:

- Any CJUMP immediately followed by its false label we let alone (there will be many of these).
- For any CJUMP followed by its true label, we switch the true and false labels and negate the condition.
- For any CJUMP($cond, a, b, l_t, l_f$) followed by neither label, we invent a new false label l'_f and rewrite the single CJUMP statement as three statements, just to achieve the condition that the CJUMP is followed by its false label:

$\text{CJUMP}(cond, a, b, l_t, l'_f)$
LABEL l'_f
JUMP(NAME l_f)

7 CH 9 Instruction Selection

Highlighting done in book's pdf.

8 CH 10 Liveness Analysis

Highlighting done in book's pdf.

9 CH 11 Register Allocation

Everything in class notes

10 CH 13 Garbage Collection

Everything in class notes