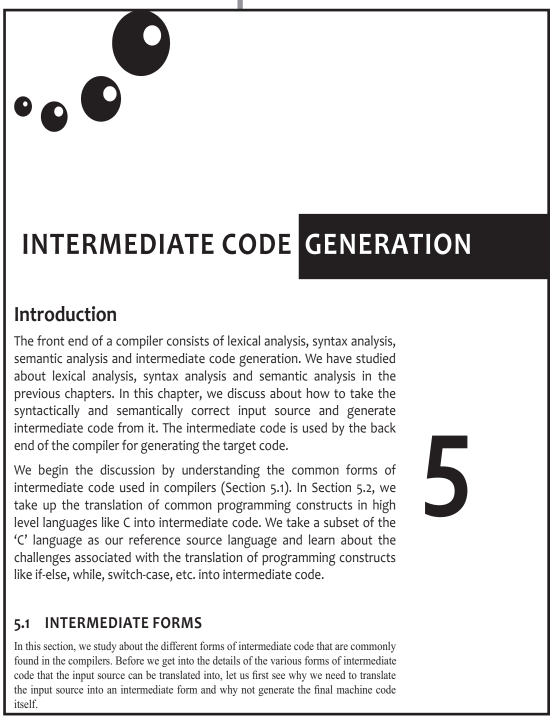
VISUAL WALKTHROUGH

INTRODUCTION

Each chapter begins with an Introduction that gives a summary of the background and the organisation of chapter's contents.



INTERMEDIATE CODE GENERATION

5

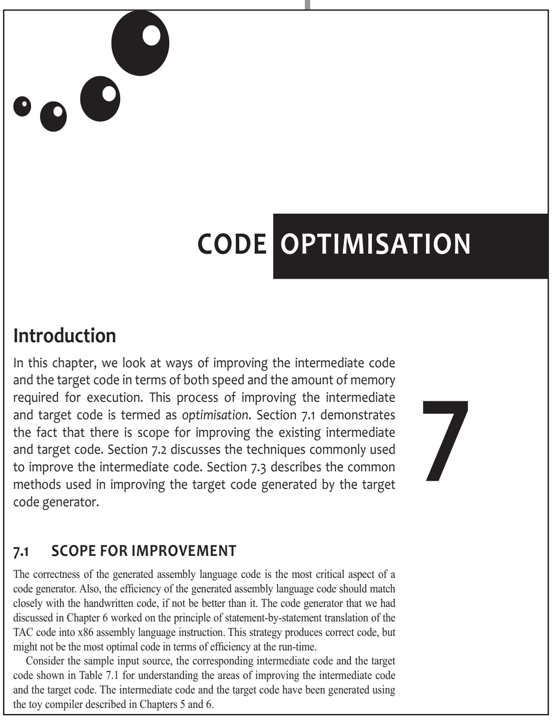
Introduction

The front end of a compiler consists of lexical analysis, syntax analysis, semantic analysis and intermediate code generation. We have studied about lexical analysis, syntax analysis and semantic analysis in the previous chapters. In this chapter, we discuss about how to take the syntactically and semantically correct input source and generate intermediate code from it. The intermediate code is used by the back end of the compiler for generating the target code.

We begin the discussion by understanding the common forms of intermediate code used in compilers (Section 5.1). In Section 5.2, we take up the translation of common programming constructs in high level languages like C into intermediate code. We take a subset of the 'C' language as our reference source language and learn about the challenges associated with the translation of programming constructs like if-else, while, switch-case, etc. into intermediate code.

5.1 INTERMEDIATE FORMS

In this section, we study about the different forms of intermediate code that are commonly found in the compilers. Before we get into the details of the various forms of intermediate code that the input source can be translated into, let us first see why we need to translate the input source into an intermediate form and why not generate the final machine code itself.



CODE OPTIMISATION

7

Introduction

In this chapter, we look at ways of improving the intermediate code and the target code in terms of both speed and the amount of memory required for execution. This process of improving the intermediate and target code is termed as *optimisation*. Section 7.1 demonstrates the fact that there is scope for improving the existing intermediate and target code. Section 7.2 discusses the techniques commonly used to improve the intermediate code. Section 7.3 describes the common methods used in improving the target code generated by the target code generator.

7.1 SCOPE FOR IMPROVEMENT

The correctness of the generated assembly language code is the most critical aspect of a code generator. Also, the efficiency of the generated assembly language code should match closely with the handwritten code, if not be better than it. The code generator that we had discussed in Chapter 6 worked on the principle of statement-by-statement translation of the TAC code into x86 assembly language instruction. This strategy produces correct code, but might not be the most optimal code in terms of efficiency at the run-time.

Consider the sample input source, the corresponding intermediate code and the target code shown in Table 7.1 for understanding the areas of improving the intermediate code and the target code. The intermediate code and the target code have been generated using the toy compiler described in Chapters 5 and 6.

7.2 INTERMEDIATE CODE OPTIMISATION

The intermediate code generated by translation scheme described in Chapter 5, is adequate in terms of correctness with respect to the input program. We saw in the previous section, that there is scope for improving the efficiency of the generated intermediate code in terms of speed of execution and size in memory. In the intermediate code optimisation phase (refer Fig. 1.9), the compiler makes a pass over the generated intermediate code and transforms it into an improved (optimised) form, which is more efficient in terms of speed and size. The transformed intermediate code is then fed to the target code generator for the generation of the target code. In the discussion in Section 7.2.1, we take a look at some of the common transformations made in the intermediate code optimisation phase of the compiler to improve the intermediate code.

7.2.1 Common Sub-expression Elimination

Consider the input source and the corresponding intermediate code in TAC format in Table 7.2. The TAC was generated from the translation scheme explained in Chapter 5. We call the intermediate code shown in Table 7.2 as ***unoptimised intermediate code*** to differentiate it from the version of intermediate code after optimisation using transformations.

Table 7.2 *Input source and the intermediate code*

Input Source	TAC

`int sum_n,sum_n2,sum_n3;`
`int sum(int n)`
`{`
`sum_n = ((n) *(n + 1))/2;`
`sum_n2=((n)*(n + 1)*(2*n + 1))/6;`
`sum_n3=(((n)*(n + 1))/2)*(((n)*(n + 1))/2);`
`}`
	`(0) proc_begin sum`
	`(1) _t0 := n + 1`
	`(2) _t1 := n * _t0`
	`(3) _t2 := _t1 / 2`
	`(4) sum_n := _t2`
	`(5) _t3 := n + 1`
	`(6) _t4 := n * _t3`
	`(7) _t5 := 2 * n`
	`(8) _t6 := _t5 + 1`
	`(9) _t7 := _t4 * _t6`
	`(10) _t8 := _t7 / 6`
	`(11) sum_n2 := _t8`
	`(12) _t9 := n + 1`
	`(13) _t10 := n * _t9`
	`(14) _t11 := _t10 / 2`
	`(15) _t12 := n + 1`
	`(16) _t13 := n * _t12`
	`(17) _t14 := _t13 / 2`
	`(18) _t15 := _t11 * _t14`
	`(19) sum_n3 := _t15`
	`(20) label .L0`
	`(21) proc_end sum`

A detailed look at the intermediate code generated in Table 7.2 indicates that the computations made in quads (1) through (3), (12) through (14) and (15) through (17) are essentially the same. These chunks of intermediate code compute the value of the common sub-expression $((n) *(n + 1))/2$, which is used in all the three summations. If we look further, the common sub-expression $((n) *(n + 1))$ is computed 4 times in the statements {1,2 }, {5,6 }, {12,13}, {15,16}. It is possible to optimise the intermediate code to have common sub-expressions computed only once in the function and then re-use the computed values at the second instance.

SECTIONS AND SUB-SECTIONS

Neatly divided into sections and sub-sections, the subject matter can be studied in a logical progression of ideas and concepts.

Table 5.1 *Input C-statements and the translated TAC*

Input C statement	TAC statements	Comments

`a = b - c + d ;`	`_t1 := b - c`
	`_t2 := _t1 + d`
	`a := _t2 ;`
		_t1 and _t2 are compiler generated temporaries. Note that one C statement is transformed into multiple TAC statements

`p_new = p + ( ( p * n * r ) /100 )`	`_t1 := p * n`
	`_t2 := _t1 * r`
	`_t3 = _t2 / 100`
	`p_new = p + _t3`
		_t1,_t2 and _t3 are compiler generated temporaries. Note that one C statement is transformed into multiple TAC statements

The number of allowable operators (like ADD, SUB, etc.) is an important factor in the design of an intermediate representation like three address code. One end of the spectrum is a restricted operator set, which allows for easy portability to multiple architectures. A restricted feature set would mean that the front end would generate a long list of TAC instructions, forcing the optimiser and code generator to do the bulk of work. At the other end of the spectrum is a feature rich operator set in the intermediate language that allows one to take advantage of an advanced processor, but is difficult to port on to low-end processors. The usual approach is to have a minimum set of allowable operators in Intermediate language, whose equivalent machine language statements would be invariably available on any processor.

The following table shows a complete list of TAC operators that we would be using in this book.

Table 5.2 *TAC operators*

#	TAC operator	Sample TAC instruction	Textual representation	Description
1	ASSIGN	ASSIGN | y | | x	$x := y$	x gets assigned the result of y op z
2	ADD	ADD | y | z | x	$x := y + z$	x gets assigned the result of y added to z
3	MUL	MUL | y | z | x	$x = y * z$	x gets assigned the result of y multiplied by z
4	DIV	DIV | y | z | x	$x := y / z$	x gets assigned the result of y divided by z
5	SUB	SUB | y | z | x	$x := y - z$	x gets assigned the result of y minus z
6	UMINUS	UMINUS | y | | x	$x := - y$	x gets assigned the value of $-y$
7	L_INDEX_ASSIGN	L_INDEX_ ASSIGN | y | i | x	$x[i] := y$	$x[i]$ denotes the content of a location which is i memory units away from the pointer contained in x. $x[i]$ gets assigned the value of y.

TABLES

Tables are provided in each chapter to aid in understanding of the text material.

The regular expression ***'bo*s'*** matches any input line where ***b*** is followed by zero or more ***o*** and then a ***s*** like say ***boos***t, ***bos***s, la***bs.***

The regular expression ***'bo+s'*** matches any input line where ***b*** is followed by one or more ***o*** and then a ***s*** like say ***boos***t, ***bos***s.

The regular expression ***'bo?s'*** matches any input line where ***b*** is followed by zero or one ***o*** and then a ***s*** like say ***bos***s, la***bs.***

```
# Matches b followed by zero or any number of o then s
$ egrep -n -e 'bo*s' my_input
6:boost
10:boss
11:labs
# Matches b followed by one or any number of o then s
$ egrep -n -e 'bo+s' my_input
6:boost
10:boss
# Matches b followed by one or no o then s
$ egrep -n -e 'bo?s' my_input
10:boss
11:labs
# Matches b followed by 1 or 2 instances of o followed by s
$ egrep -n -e 'bo{1,2}s' my_input
6:boost
10:boss
```

Comments for Reader's Understanding in *Italics*

The regular expression ***'oa|or'*** matches any input line where ***oa*** or ***ort*** exists like say *goat, port*ed.

```
# Matches oa or ort
$ egrep -n -e 'oa|ort' my_input
5:goat
8:ported
```

User Commands in **Bold**

The regular expression 'The regular expression ***' (oa|os)t'*** matches any input line where ***oa*** or ***os*** followed by ***t*** exists like say ***goat,*** bo***ost.***

```
# Matches oa or os followed by t
$ egrep -n -e '(oa|os)t' my_input
5:goat
6:boost
```

Having understood the concepts of regular expressions, let's define the constructs of C language like identifiers, constants, etc. using regular expressions.

The C language keywords are the easiest ones to be described using regular expression. The regular expression is same as the keyword. For example, the keyword 'goto' is represented by a regular expression '*goto*', and 'switch' by regular expression '*switch*', and so on.

A C identifier begins with an alphabet or underscore, followed by either an alphabet or underscore or digit. A C identifier is represented by a regular expression `'[a-zA-Z_]([a-zA-Z_]|[0-9])*'`

DIALOGS

Dialogs showing the human-computer interactions are provided in each chapter to help the readers appreciate the relevant practical aspects.

The output from the computer in regular font.

1.1 THE BIGGER PICTURE

A compiler works in tandem with a few other utilities like preprocessor, assembler, linker and so on to produce binaries that can be executed. In this section we look at the process of generating an executable binary from a sample input program written in C and understand how these utilities fit in.

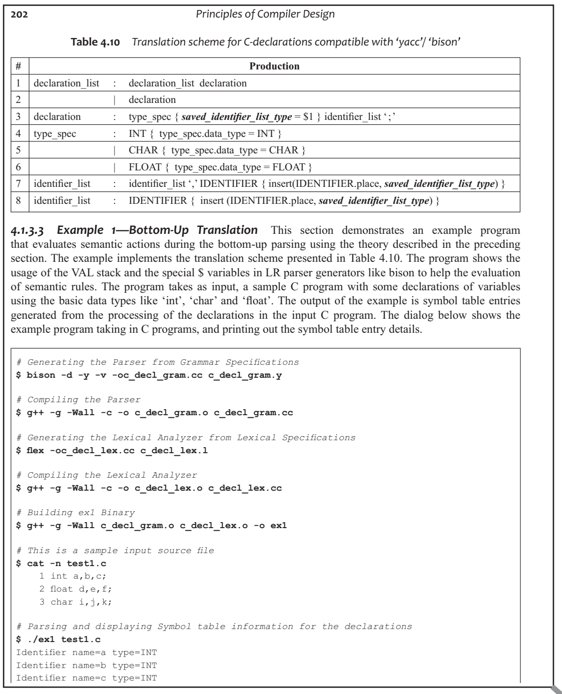
Let's start off with a sample C program (ex1.c) and see how we transform it into a binary that can be executed. The following dialog shows the sample C program (ex1.c) being transformed into an executable using GNU's compiler collection (*gcc*)—a freely available compiler collection.

```
# A Sample input C file
$ cat -n ex1.c
     1
     2  #define SUCCESS 0
     3
     4  /* The function prototype for printf found in stdio.h */
     5  extern int printf(const char *, ...);
     6
     7  int main()
     8  {
     9         printf("Hello World\n");
    10
    11         /* returning 0 to the Operating system */
    12         return(SUCCESS);
    13  }

# Creating an executable from sample input file using the GNU C compiler system
$ gcc -Wall ex1.c -o ex1

# Invoking the executable
$ ./ex1
Hello World
```

From the above dialog, we can understand that the GNU compiler collection (gcc) binary is invoked with the input C source file as the argument for compilation. This outputs an executable ex1, which can be invoked on the command line to get the desired effect. The above dialog abstracts us from a lot of behind-the-scene activity that is involved in conversion of the input C file into executable.

Let's try out the compilation of the same C program (ex1.c), this time with extra options to 'gcc' in order to know all the utilities that get involved in the transformation of ex1.c to an executable binary. The following dialog shows the compilation of ex1.c with gcc using the extra options for getting a detailed account of the compilation.

```
# Compiling with verbose option and preserving the intermediate files
$ gcc --save-temps --verbose ex1.c -o ex1
Reading specs from /usr/lib/gcc-lib/i686-pc-cygwin/2.95.3-5/specs
gcc version 2.95.3-5 (cygwin special)
/usr/lib/gcc-lib/i686-pc-cygwin/2.95.3-5/cpp0.exe -lang-c -v -D__GNUC__=2 -D__GNUC_MINOR__=95 -
D_X86_=1 -D_X86_=1 -Asystem(winnt) -Acpu(i386) -Amachine(i386) -Di386 -D__i386 -D__i386__ -Di686 -
Dpentiumpro -D__i686 -D__i686__ -D__pentiumpro -D__pentiumpro__ -
D__stdcall=__attribute__((__stdcall_)) -D__cdecl=__attribute__((__cdecl__)) -
D_stdcall=__attribute__((__stdcall__)) -D_cdecl=__attribute__((__cdecl__)) -
D__declspec(x)=__attribute__((x)) -D__CYGWIN32__ -D__CYGWIN__ -Dunix -D__unix__ -D__unix -isystem/
usr/local/include -idirafter /usr/include -idirafter /usr/include/w32api ex1.c ex1.i
```



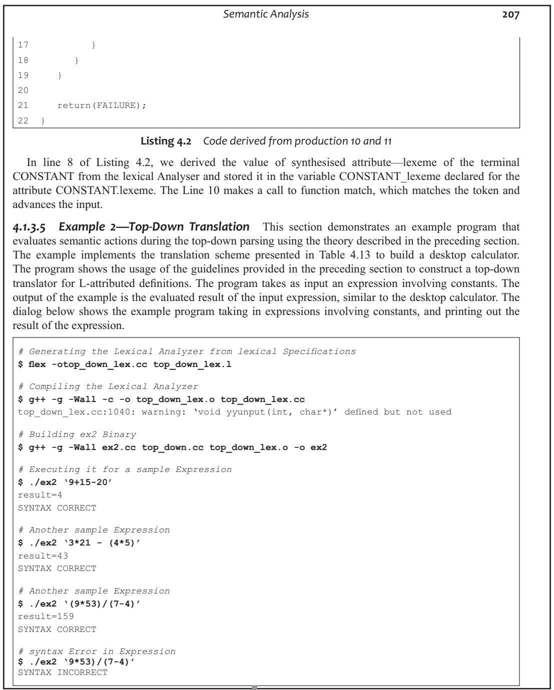
202 *Principles of Compiler Design*

Table 4.10 *Translation scheme for C-declarations compatible with 'yacc'/'bison'*

#			Production
1	declaration_list	:	declaration_list declaration
2		|	declaration
3	declaration	:	type_spec { ***saved_identifier_list_type*** = $1 } identifier_list ';'
4	type_spec	:	INT { type_spec.data_type = INT }
5		|	CHAR { type_spec.data_type = CHAR }
6		|	FLOAT { type_spec.data_type = FLOAT }
7	identifier_list	:	identifier_list ',' IDENTIFIER { insert(IDENTIFIER.place, ***saved_identifier_list_type***) }
8	identifier_list	:	IDENTIFIER { insert (IDENTIFIER.place, ***saved_identifier_list_type***) }

4.1.3.3 Example 1—Bottom-Up Translation This section demonstrates an example program that evaluates semantic actions during the bottom-up parsing using the theory described in the preceding section. The example implements the translation scheme presented in Table 4.10. The program shows the usage of the VAL stack and the special $ variables in LR parser generators like bison to help the evaluation of semantic rules. The program takes as input, a sample C program with some declarations of variables using the basic data types like 'int', 'char' and 'float'. The output of the example is symbol table entries generated from the processing of the declarations in the input C program. The dialog below shows the example program taking in C programs, and printing out the symbol table entry details.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v -oc_decl_gram.cc c_decl_gram.y

# Compiling the Parser
$ g++ -g -Wall -c -o c_decl_gram.o c_decl_gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex -oc_decl_lex.cc c_decl_lex.l

# Compiling the Lexical Analyzer
$ g++ -g -Wall -c -o c_decl_lex.o c_decl_lex.cc

# Building ex1 Binary
$ g++ -g -Wall c_decl_gram.o c_decl_lex.o -o ex1

# This is a sample input source file
$ cat -n test1.c
    1 int a,b,c;
    2 float d,e,f;
    3 char i,j,k;

# Parsing and displaying Symbol table information for the declarations
$ ./ex1 test1.c
Identifier name=a type=INT
Identifier name=b type=INT
Identifier name=c type=INT
```



Semantic Analysis 207

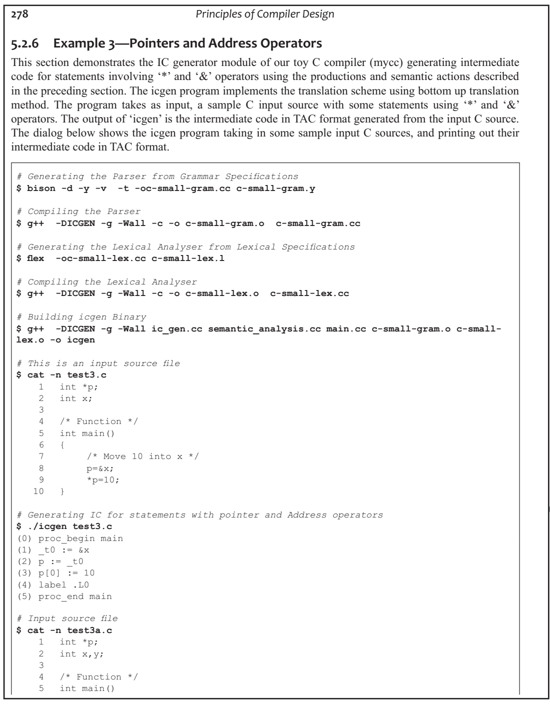
```
17           }
18        }
19    }
20
21    return(FAILURE);
22 }
```

Listing 4.2 *Code derived from production 10 and 11*

In line 8 of Listing 4.2, we derived the value of synthesised attribute—lexeme of the terminal CONSTANT from the lexical Analyser and stored it in the variable CONSTANT_lexeme declared for the attribute CONSTANT.lexeme. The Line 10 makes a call to function match, which matches the token and advances the input.

4.1.3.5 Example 2—Top-Down Translation This section demonstrates an example program that evaluates semantic actions during the top-down parsing using the theory described in the preceding section. The example implements the translation scheme presented in Table 4.13 to build a desktop calculator. The program shows the usage of the guidelines provided in the preceding section to construct a top-down translator for L-attributed definitions. The program takes as input an expression involving constants. The output of the example is the evaluated result of the input expression, similar to the desktop calculator. The dialog below shows the example program taking in expressions involving constants, and printing out the result of the expression.

```
# Generating the Lexical Analyzer from lexical Specifications
$ flex -otop_down_lex.cc top_down_lex.l

# Compiling the Lexical Analyzer
$ g++ -g -Wall -c -o top_down_lex.o top_down_lex.cc
top_down_lex.cc:1040: warning: 'void yyunput(int, char*)' defined but not used

# Building ex2 Binary
$ g++ -g -Wall ex2.cc top_down.cc top_down_lex.o -o ex2

# Executing it for a sample Expression
$ ./ex2 '9+15-20'
result=4
SYNTAX CORRECT

# Another sample Expression
$ ./ex2 '3*21 - (4*5)'
result=43
SYNTAX CORRECT

# Another sample Expression
$ ./ex2 '(9*53)/(7-4)'
result=159
SYNTAX CORRECT

# syntax Error in Expression
$ ./ex2 '9*53)/(7-4)'
SYNTAX INCORRECT
```



278 *Principles of Compiler Design*

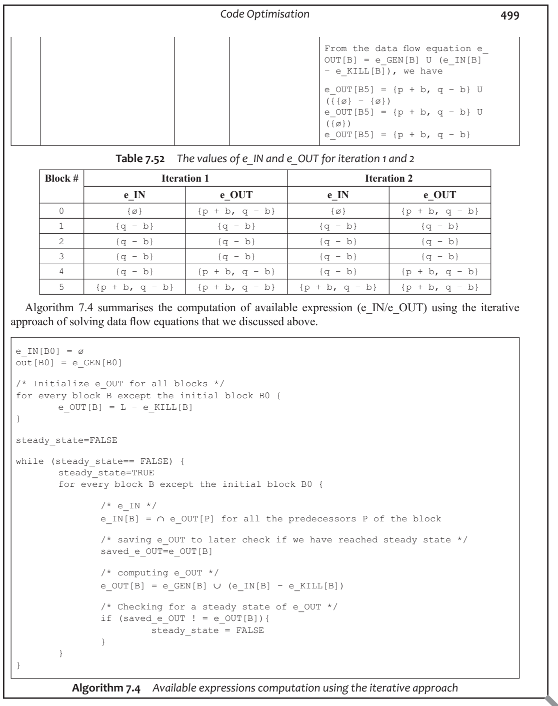
5.2.6 Example 3—Pointers and Address Operators

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for statements involving '*' and '&' operators using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom up translation method. The program takes as input, a sample C input source with some statements using '*' and '&' operators. The output of 'icgen' is the intermediate code in TAC format generated from the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++  -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# This is an input source file
$ cat -n test3.c
     1   int *p;
     2   int x;
     3
     4   /* Function */
     5   int main()
     6   {
     7        /* Move 10 into x */
     8        p=&x;
     9        *p=10;
    10   }

# Generating IC for statements with pointer and Address operators
$ ./icgen test3.c
(0) proc_begin main
(1) _t0 := &x
(2) p := _t0
(3) p[0] := 10
(4) label .L0
(5) proc_end main

# Input source file
$ cat -n test3a.c
     1   int *p;
     2   int x,y;
     3
     4   /* Function */
     5   int main()
```

SAMPLE CODE IMPLEMENTATION

A Dialog pertaining to a sample implementation immediately follows the theory to reinforce the ideas correctly. All the source code used in the textbook is online and can be downloaded from the website http://www.mhhe.com/raghavan/pcd.



Code Optimisation **499**

				From the data flow equation e_OUT[B] = e_GEN[B] U (e_IN[B] – e_KILL[B]), we have e_OUT[B5] = {p + b, q – b} U ({{ø} – {ø}) e_OUT[B5] = {p + b, q – b} U ({ø}) e_OUT[B5] = {p + b, q – b}

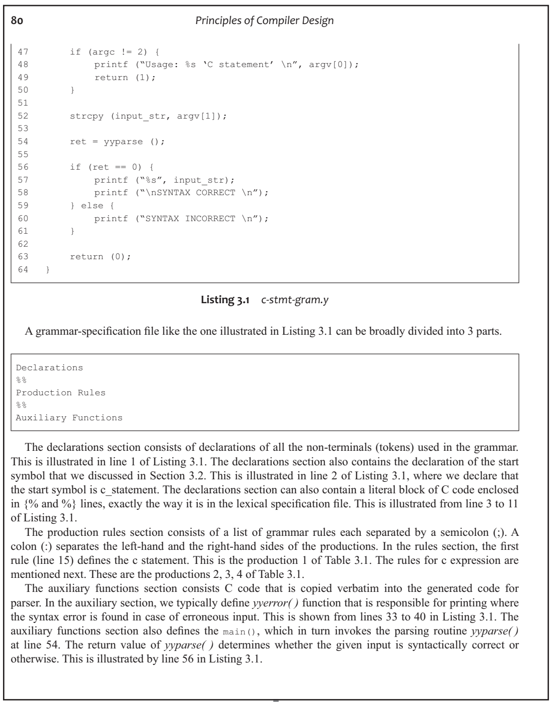
Table 7.52 *The values of e_IN and e_OUT for iteration 1 and 2*

Block #	Iteration 1		Iteration 2	
	e_IN	e_OUT	e_IN	e_OUT
0	{ø}	{p + b, q – b}	{ø}	{p + b, q – b}
1	{q – b}	{q – b}	{q – b}	{q – b}
2	{q – b}	{q – b}	{q – b}	{q – b}
3	{q – b}	{q – b}	{q – b}	{q – b}
4	{q – b}	{p + b, q – b}	{q – b}	{p + b, q – b}
5	{p + b, q – b}	{p + b, q – b}	{p + b, q – b}	{p + b, q – b}

Algorithm 7.4 summarises the computation of available expression (e_IN/e_OUT) using the iterative approach of solving data flow equations that we discussed above.

```
e_IN[B0] = ø
out[B0] = e_GEN[B0]

/* Initialize e_OUT for all blocks */
for every block B except the initial block B0 {
        e_OUT[B] = L – e_KILL[B]
}

steady_state=FALSE

while (steady_state== FALSE) {
        steady_state=TRUE
        for every block B except the initial block B0 {

                /* e_IN */
                e_IN[B] = ∩ e_OUT[P] for all the predecessors P of the block

                /* saving e_OUT to later check if we have reached steady state */
                saved_e_OUT=e_OUT[B]

                /* computing e_OUT */
                e_OUT[B] = e_GEN[B] ∪ (e_IN[B] – e_KILL[B])

                /* Checking for a steady state of e_OUT */
                if (saved_e_OUT ! = e_OUT[B]){
                         steady_state = FALSE
                }
        }
}
```

Algorithm 7.4 *Available expressions computation using the iterative approach*



80 *Principles of Compiler Design*

```
47       if (argc != 2) {
48           printf ("Usage: %s 'C statement' \n", argv[0]);
49           return (1);
50       }
51
52       strcpy (input_str, argv[1]);
53
54       ret = yyparse ();
55
56       if (ret == 0) {
57           printf ("%s", input_str);
58           printf ("\nSYNTAX CORRECT \n");
59       } else {
60           printf ("SYNTAX INCORRECT \n");
61       }
62
63       return (0);
64   }
```

Listing 3.1 *c-stmt-gram.y*

A grammar-specification file like the one illustrated in Listing 3.1 can be broadly divided into 3 parts.

```
Declarations
%%
Production Rules
%%
Auxiliary Functions
```

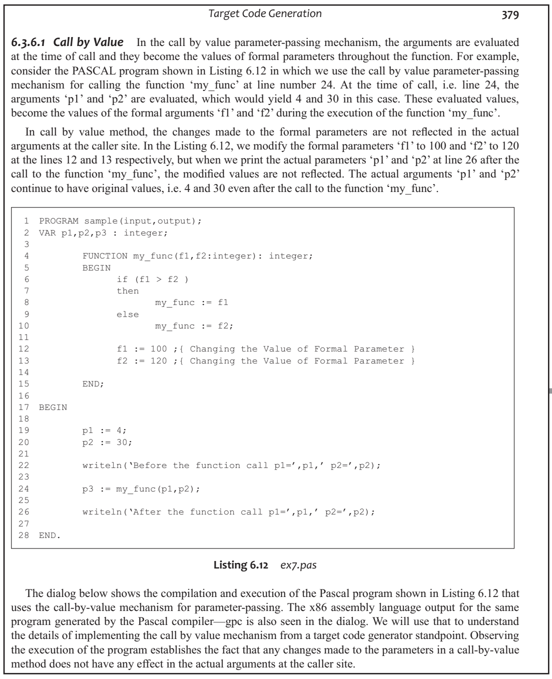
The declarations section consists of declarations of all the non-terminals (tokens) used in the grammar. This is illustrated in line 1 of Listing 3.1. The declarations section also contains the declaration of the start symbol that we discussed in Section 3.2. This is illustrated in line 2 of Listing 3.1, where we declare that the start symbol is c_statement. The declarations section can also contain a literal block of C code enclosed in {% and %} lines, exactly the way it is in the lexical specification file. This is illustrated from line 3 to 11 of Listing 3.1.

The production rules section consists of a list of grammar rules each separated by a semicolon (;). A colon (:) separates the left-hand and the right-hand sides of the productions. In the rules section, the first rule (line 15) defines the c statement. This is the production 1 of Table 3.1. The rules for c expression are mentioned next. These are the productions 2, 3, 4 of Table 3.1.

The auxiliary functions section consists C code that is copied verbatim into the generated code for parser. In the auxiliary section, we typically define *yyerror( )* function that is responsible for printing where the syntax error is found in case of erroneous input. This is shown from lines 33 to 40 in Listing 3.1. The auxiliary functions section also defines the `main()`, which in turn invokes the parsing routine *yyparse( )* at line 54. The return value of *yyparse( )* determines whether the given input is syntactically correct or otherwise. This is illustrated by line 56 in Listing 3.1.

LISTINGS AND ALGORITHMS

Code Listings and Algorithm Specifications have been provided at appropriate locations in the chapters.



Target Code Generation **379**

6.3.6.1 Call by Value In the call by value parameter-passing mechanism, the arguments are evaluated at the time of call and they become the values of formal parameters throughout the function. For example, consider the PASCAL program shown in Listing 6.12 in which we use the call by value parameter-passing mechanism for calling the function 'my_func' at line number 24. At the time of call, i.e. line 24, the arguments 'p1' and 'p2' are evaluated, which would yield 4 and 30 in this case. These evaluated values, become the values of the formal arguments 'f1' and 'f2' during the execution of the function 'my_func'.

In call by value method, the changes made to the formal parameters are not reflected in the actual arguments at the caller site. In the Listing 6.12, we modify the formal parameters 'f1' to 100 and 'f2' to 120 at the lines 12 and 13 respectively, but when we print the actual parameters 'p1' and 'p2' at line 26 after the call to the function 'my_func', the modified values are not reflected. The actual arguments 'p1' and 'p2' continue to have original values, i.e. 4 and 30 even after the call to the function 'my_func'.

```
 1  PROGRAM sample(input,output);
 2  VAR p1,p2,p3 : integer;
 3
 4          FUNCTION my_func(f1,f2:integer): integer;
 5          BEGIN
 6                  if (f1 > f2 )
 7                  then
 8                          my_func := f1
 9                  else
10                          my_func := f2;
11
12                  f1 := 100 ;{ Changing the Value of Formal Parameter }
13                  f2 := 120 ;{ Changing the Value of Formal Parameter }
14
15          END;
16
17  BEGIN
18
19          p1 := 4;
20          p2 := 30;
21
22          writeln('Before the function call p1=',p1,' p2=',p2);
23
24          p3 := my_func(p1,p2);
25
26          writeln('After the function call p1=',p1,' p2=',p2);
27
28  END.
```

Listing 6.12 *ex7.pas*

The dialog below shows the compilation and execution of the Pascal program shown in Listing 6.12 that uses the call-by-value mechanism for parameter-passing. The x86 assembly language output for the same program generated by the Pascal compiler—gpc is also seen in the dialog. We will use that to understand the details of implementing the call by value mechanism from a target code generator standpoint. Observing the execution of the program establishes the fact that any changes made to the parameters in a call-by-value method does not have any effect in the actual arguments at the caller site.

```
               c :=40;
               writeln('Value of c is ',c); (* c is 40 here *)
               P2();
               writeln('Value of c is ',c); (* c is 25 here *)
          END;
     BEGIN
          P1();
     END.
```

Listing 6.8 A Pascal program with nested procedures

Programming languages like LISP and APL allow variables to be bound to a storage depending on the current activations. In these cases the variable can be resolved to the appropriate declaration only at the run-time depending on the current activations. In order to implement such dynamic scoping, it is necessary to keep track of the chain of the current activations. The optional ***control link*** in an activation record helps in maintaining a track of the current activations and implementing dynamic scope. Following the control link of the current activation record, we can make a chain of all the functions that are currently active. This helps in implementing the dynamic scope.

The activation record contains a field for storing the return value of a function. The callee stores the return value in this field before returning the control to the caller. The caller copies it from this field into the appropriate variable as defined in the source program. In practice, many of the compilers, have the return value and the arguments passed in registers, whenever feasible rather than having them as a part of activation record. The register access is faster than memory access and hence passing the return values and arguments in registers is more efficient.

Activation records are allocated space in the stack Area in C run-time environment. The Old FORTRAN77 compilers used the static area for housing the activation records. The run-time environments for functional languages like LISP allocate space for activation records on the heap.

6.3.4.1 Activation Record in C Run-time Environment In C runtime environment, the activation records are allocated storage space on the stack. When a procedure is called, a new activation record is pushed on to the stack. When the procedure is complete, the activation record is popped-out of the stack. The top of the stack is usually pointed to by a register called SP (stack pointer). An activation record can be allocated by moving SP with an amount equal to the size of activation record. The activation record is de-allocated by moving the SP back by an amount equal to the size of activation record. For example, consider the activation of a function 'my_func()' having an activation record of size, say, 40 bytes. The SP is moved (decremented in this case) by 40 bytes to allocate an activation record for my_func(). The SP is moved back (incremented by 40) to de-allocate the activation record for my_func() after the execution of my_func() is complete. Figure 6.16 shows the run-time stack, before, during and after the activation of my_func().

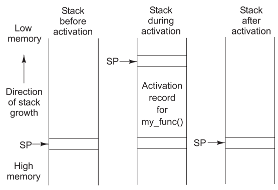


Fig. 6.16 *Allocating and de-allocating space for activation records*

system-wide start up object file (crt0.o) and makes an executable. The linker also links the ex1.o file with other system-wide libraries, including the C library containing the function definitions for printf, scanf, etc. The libraries that are used by the linker are the ones given by –l option during the invocation of linker. The output of the linker is an executable (ex1.exe) that can be invoked on the command line. The dialog below shows us that the final executable ex1.exe is a MS Windows binary for Intel 80386, which can be invoked on the console.

```
# The properties of the executable ex1.exe
$ file ex1.exe
ex1.exe: MS Windows PE Intel 80386 console executable not relocatable
```

The whole process of transforming an input C source file into an executable binary is summarised in Fig. 1.1.

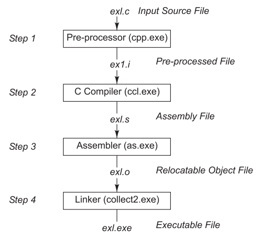


Fig. 1.1 *Transforming an input C-source file into an executable*

Even though Fig. 1.1 shows the transformation of an input source file written in C language into an executable form, the steps are similar for other compiled languages also.

1.2 THE COMPILER

The main focus of the book is to understand the details of working of a compiler, i.e. the step2 of Fig. 1.1. The compiler takes the pre-processed file as the input and translates it into an equivalent assembly language file. In this section, we will get an overview of how a compiler translates a pre-processed input file into an assembly language file.

The translation of the input source (pre-processed file) into target assembly language file can be divided into two stages called as ***front end*** (or analysis) and ***back end*** (or synthesis).

The front end of the compiler transforms the input source into intermediate code. The intermediate code (sometimes called intermediate representation—IR) is a machine-independent representation of the input source program.

FIGURES

Figures are used exhaustively in the text to illustrate the concepts and methods described.

Consider a monolithic compiler for C language that generates machine instructions directly from the input source for an 80×86 processor system. Let's say it needs to be modified to generate machine instructions for SPARC processor system. The effort involved in modifying the 80×86-based compiler for re-targeting to SPARC platform is high. It requires the intricate knowledge of the machine instructions of both the 80×86 system as well as SPARC System. Also, the translation to final machine code from the input source language makes the generation of optimal code difficult because it would not have the context of the entire program.

Consider another compiler that is broken into modular elements called as front end and the back end, as explained in Chapter 1. The re-targeting of such a compiler from 80×86 to SPARC system is illustrated in Fig. 5.1. The front end of the compiler for a source language remains same irrespective of the machine code generated. The output of the front end of the compiler is an intermediate form that does not depend on the specifics of the processor. The back end of the compiler converts the intermediate code into the respective machine instructions as required. This approach allows the re-use of a large portion of the compiler without modification during the re-targeting to a different processor.

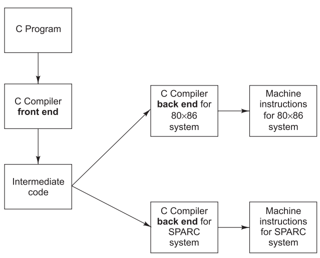


Fig. 5.1 *Retargeting of a compiler*

Some of the advantages in this approach of breaking up the compiler into front end and back end are:

1. It is easy to re-target the compiler to generate code for newer and different processors. As seen in the discussion previously, the re-targeting of the compiler could be highly effort intensive but for the presence of intermediate code.
2. The compiler can be easily extended to support an additional input source language by adding the required front end and retaining the same back end.
3. It facilitates machine independent code optimisation. The intermediate code generated by the front end can be optimised by using several specialised techniques. This optimisation is different from the target code optimisation that can be done during the code generation for the actual processor by the back end system.

Most of the modern compilers take this approach of partitioning the job of the compiler into front end and back end.

Lexical Analysis **19**

Comments and white space (like tab, blank, new line) do not influence code generation. The lexical analyser strips out the comments and white space in the source program. For example, in Fig. 2.1, the lexical analyser stripped out the white space (line 5), comment (line 6) of the input C program and did not return them as tokens.

The part of the input stream that qualifies for a certain type of token is called as ***lexeme***. For example, in line 4 of the input the letters 'int' qualifies for a keyword in C language. 'int' is called as lexeme in this case. The other lexemes shown in Fig. 2.1 are 'main' (token type is identifier), 'for' (token type is keyword), etc.

The lexical analyser keeps track of the new line characters, so that it can output the line number with associated error messages, in case of errors in the input source program. This is extremely useful for the programmer to correct syntax errors. For example, consider the C program shown in the dialog below in which the line 5 does not end with a semicolon (;). On trying to compile it using *GNU C compiler*, the following output was observed:

```
# An input C program. A semicolon (;) is missing in Line 5
$ cat -n test1.c
     1  #include <stdio.h>
     2
     3  int main()
     4  {
     5       printf ("Hello World \n")
     6       return(0);
     7  }

# Compiling the C program
$ gcc test1.c -o test1
test1.c: In function 'main':
test1.c:6: error: parse error before "return"
```

The error message in the dialog indicates that a parse error was encountered on line 6, before the token 'return'. This message indicating the line number was possible because the lexical analyser kept a count of the number of new lines that it has encountered till that point of the source program.

The lexical analyser in conjunction with the parser is responsible for creating ***symbol table***, a data structure containing information that is used in various stages of the compiler. The symbol table consists of entries describing various identifiers used in the source program. Typically, each entry in the symbol table consists of the lexeme of the identifier and all the attributes associated with it. While some of the attributes pertaining to an entry are filled in at lexical analyser/parser level, the other attributes in the entry would be progressively filled by subsequent stages of compilation. As an

Syntax Analysis **85**

```
     7
     8          var1 = 0;
     9          var2 = 10;
    10
    11          printf("This is message 1 ")
    12
    13          var1 = var2 ;
    14
    15          for( i = var1; i < var2; i++){
    16               printf("This is iteration %d ",i);
    17          }
    18  }
```

The input C source program *test1.c* has two errors. (1) There is a missing semicolon in line 11 and (2) the variable 'i' used in line 15 has not been declared earlier.

The dialog below shows how the GNU's C compiler 'gcc' parses the above program.

```
$ gcc test1.c -o test1
test1.c: In function 'main':
test1.c:13: parse error before 'var1'
test1.c:15: 'i' undeclared (first use in this function)
test1.c:15: (Each undeclared identifier is reported only once
test1.c:15: for each function it appears in.)
```

The parser in gcc has reported the error in line 13 before the variable 'var1', which is nothing but the end of line 11. This is indicative of missing semicolon in line 11. Note that the parser of *gcc* did not stop there, it continued parsing the subsequent lines of input source program and identified an error in line number 15. The parser in *gcc* has performed error recovery from earlier error in line 13 and continued parsing. The error reporting on line number 15 clearly says that 'i' is not declared. Note that, the parser was smart enough to report the non-declaration of 'i' once, despite being used more than once.

The above example demonstrates the error reporting and error recovery features of a parser.

The main considerations in error reporting are:

- The error handler should report the place in the input source program, where the error has occurred. The offending line number should be emitted for the programmer to correct the mistake.
- The diagnostic message emitted out by the error handler module of the parser should give out enough information to help the programmer correct the mistake in the input source program.

The job of error recovery for the error handler is trickier. The following are some of the considerations in error recovery:

- The error recovery should not be partial where spurious errors not made by the programmer are falsely identified as errors and displayed.
- The error recovery should also be cautious not to get into a bind when a totally unexpected input is given.
- The compiler designer needs to decide if error repair feature should be incorporated in the error handler. Usually error repair is not very cost-effective except in situations where the input source program is from beginners to programming.

There are several error-recovery strategies that can normally be applied in the error handler of a parser. They are:

Semantic Analysis **187**

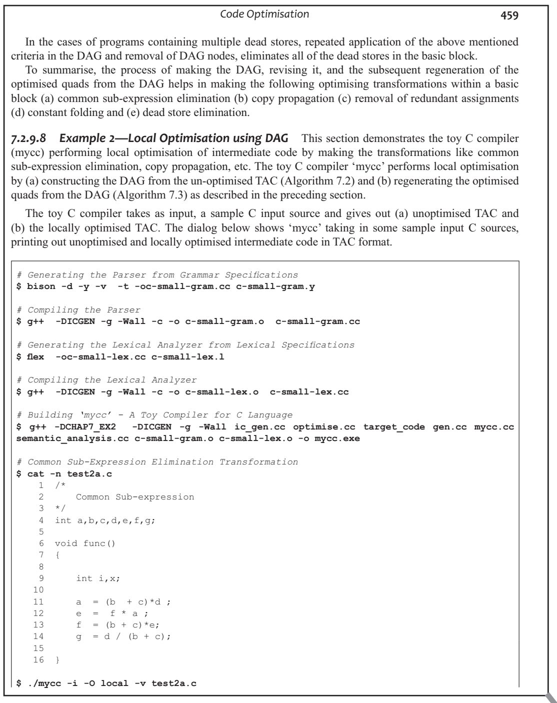
analysis is the last phase in which we reject incorrect input programs and flash error messages for the user to correct them.

The following dialog examines a few C programs, which have some semantic errors and shows us how the GNU C compiler detects and reports them. These examples give us a feel of what kinds of errors are detected in semantic analysis. Observe that all of these programs are syntactically correct, but have semantic errors.

```
# A C Program using an undeclared variable
$ cat -n sem_err1.c
     1
     2  int main()
     3  {
     4       int a,b;
     5
     6       a=1;
     7       b=2;
     8       c=3; /* Use of undeclared variable */
     9
    10       a = b + c;
    11
    12       return(a);
    13
    14  }

# The Compiler detects it and reports the error
$ gcc -Wall sem_err1.c -o sem_err1
sem_err1.c: In function `main':
sem_err1.c:8: error: `c' undeclared (first use in this function)
sem_err1.c:8: error: (Each undeclared identifier is reported only once
sem_err1.c:8: error: for each function it appears in.)

# A C Program Assigning a float to char pointer
$ cat -n sem_err2.c
     1
     2  int main()
     3  {
     4       char *a;
     5
     6       float b,c;
     7
     8       b = 30.45;
     9       c = 40.36;
    10
    11       a = b + c; /* Assigning a float to char pointer */
    12
    13       return(0);
    14
    15  }
```

PRODUCTION COMPILER REFERENCES

Dialogs exemplifying the behaviour of a production compiler suite (gcc) have been provided in the pertinent sections of the textbook.



Code Optimisation 459

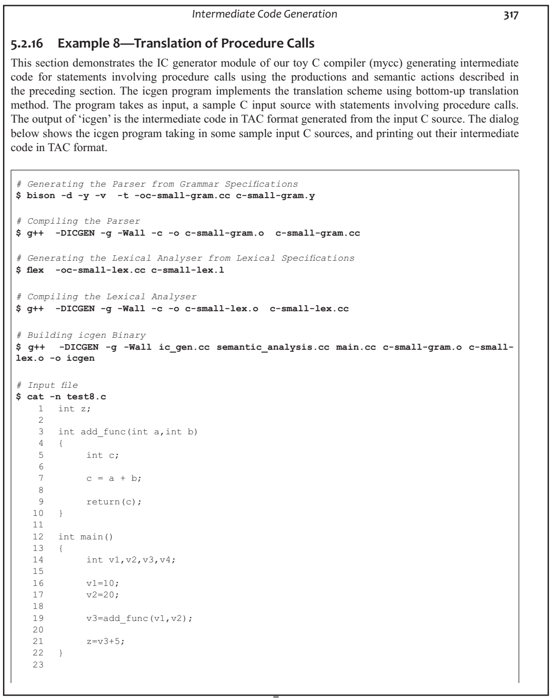
In the cases of programs containing multiple dead stores, repeated application of the above mentioned criteria in the DAG and removal of DAG nodes, eliminates all of the dead stores in the basic block.

To summarise, the process of making the DAG, revising it, and the subsequent regeneration of the optimised quads from the DAG helps in making the following optimising transformations within a basic block (a) common sub-expression elimination (b) copy propagation (c) removal of redundant assignments (d) constant folding and (e) dead store elimination.

7.2.9.8 Example 2—Local Optimisation using DAG This section demonstrates the toy C compiler (mycc) performing local optimisation of intermediate code by making the transformations like common sub-expression elimination, copy propagation, etc. The toy C compiler 'mycc' performs local optimisation by (a) constructing the DAG from the un-optimised TAC (Algorithm 7.2) and (b) regenerating the optimised quads from the DAG (Algorithm 7.3) as described in the preceding section.

The toy C compiler takes as input, a sample C input source and gives out (a) unoptimised TAC and (b) the locally optimised TAC. The dialog below shows 'mycc' taking in some sample input C sources, printing out unoptimised and locally optimised intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyzer from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyzer
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building 'mycc' - A Toy Compiler for C Language
$ g++ -DCHAP7_EX2  -DICGEN -g -Wall ic_gen.cc optimise.cc target_code gen.cc mycc.cc
semantic_analysis.cc c-small-gram.o c-small-lex.o -o mycc.exe

# Common Sub-Expression Elimination Transformation
$ cat -n test2a.c
     1  /*
     2      Common Sub-expression
     3  */
     4  int a,b,c,d,e,f,g;
     5
     6  void func()
     7  {
     8
     9      int i,x;
    10
    11      a  = (b  + c)*d ;
    12      e  =  f * a ;
    13      f  = (b + c)*e;
    14      g  = d / (b + c);
    15
    16  }

$ ./mycc -i -O local -v test2a.c
```

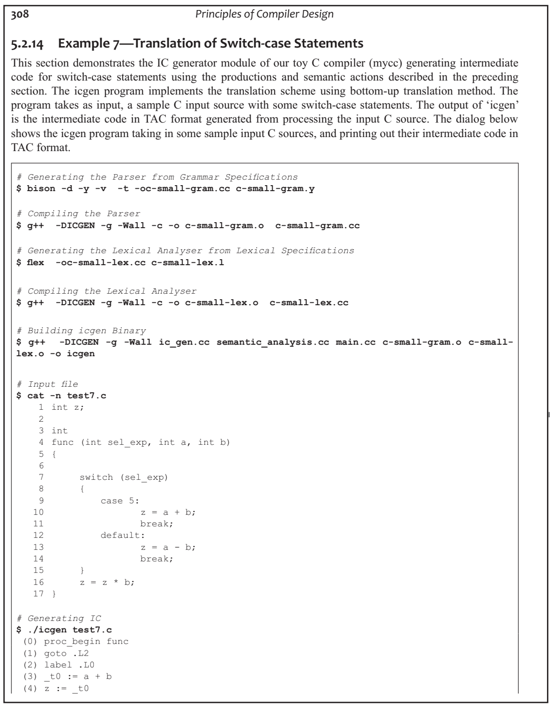


Intermediate Code Generation 317

5.2.16 Example 8—Translation of Procedure Calls

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for statements involving procedure calls using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom-up translation method. The program takes as input, a sample C input source with statements involving procedure calls. The output of 'icgen' is the intermediate code in TAC format generated from the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++  -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# Input file
$ cat -n test8.c
     1  int z;
     2
     3  int add_func(int a,int b)
     4  {
     5       int c;
     6
     7       c = a + b;
     8
     9       return(c);
    10  }
    11
    12  int main()
    13  {
    14       int v1,v2,v3,v4;
    15
    16       v1=10;
    17       v2=20;
    18
    19       v3=add_func(v1,v2);
    20
    21       z=v3+5;
    22  }
    23
```



308 Principles of Compiler Design

5.2.14 Example 7—Translation of Switch-case Statements

This section demonstrates the IC generator module of our toy C compiler (mycc) generating intermediate code for switch-case statements using the productions and semantic actions described in the preceding section. The icgen program implements the translation scheme using bottom-up translation method. The program takes as input, a sample C input source with some switch-case statements. The output of 'icgen' is the intermediate code in TAC format generated from processing the input C source. The dialog below shows the icgen program taking in some sample input C sources, and printing out their intermediate code in TAC format.

```
# Generating the Parser from Grammar Specifications
$ bison -d -y -v  -t -oc-small-gram.cc c-small-gram.y

# Compiling the Parser
$ g++  -DICGEN -g -Wall -c -o c-small-gram.o  c-small-gram.cc

# Generating the Lexical Analyser from Lexical Specifications
$ flex  -oc-small-lex.cc c-small-lex.l

# Compiling the Lexical Analyser
$ g++  -DICGEN -g -Wall -c -o c-small-lex.o  c-small-lex.cc

# Building icgen Binary
$ g++  -DICGEN -g -Wall ic_gen.cc semantic_analysis.cc main.cc c-small-gram.o c-small-
lex.o -o icgen

# Input file
$ cat -n test7.c
     1 int z;
     2
     3 int
     4 func (int sel_exp, int a, int b)
     5 {
     6
     7      switch (sel_exp)
     8      {
     9          case 5:
    10                  z = a + b;
    11                  break;
    12          default:
    13                  z = a - b;
    14                  break;
    15      }
    16      z = z * b;
    17 }

# Generating IC
$ ./icgen test7.c
 (0) proc_begin func
 (1) goto .L2
 (2) label .L0
 (3) _t0 := a + b
 (4) z := _t0
```

A TOY C COMPILER IMPLEMENTATION

A Toy C Language compiler is built progressively chapter by chapter using the concepts explained in each chapter.

following chapters, we would also familiarise ourselves with tools that would help perform the tasks in that phase easily. The examples in the chapters would illustrate the principles discussed therein.

A toy C compiler (mycc) is developed incrementally by adding the corresponding module as we progress chapter by chapter in this book. We demonstrate the capabilities of the respective module in our toy C compiler as we progress chapterwise. For example, the toy C compiler's semantic analyser module is demonstrated in Chapter 4—Semantic Analysis, the intermediate code generator module in Chapter 5—Intermediate Code Generation, and so on.

SUMMARY

A compiler is a software utility that translates code written in higher language like C or C++ into target language. The target language is usually a low-level language like assembly language or machine language. The job of the compiler can be split into two distinct stages, namely the front end and the back end. The front end is responsible for translating the input source for compilation into a form known as the Intermediate code, which is independent of the target processor architecture. The back end converts the Intermediate code into the assembly language or the machine language of the target processor. The front end and the back end can be logically divided into phases, where each phase has a specific task to accomplish. The front end can be divided into lexical analysis, syntax analysis, semantic analysis, intermediate code generation and intermediate code optimisation phases. The back end can be split into target code generation and target code optimisation phases. We shall study about each of the phases in detail in the forthcoming chapters. A compiler can be termed as a multi-pass or a single-pass compiler depending on the number of times it reads the equivalent of the entire input source in the form of tokens or parse tree or Intermediate code and likewise. The main data structures involved in a compiler implementation are symbol table, literal table and optionally, a parse tree.

REVIEW QUESTIONS AND EXERCISES

1.1 What is a compiler? What is its primary function? What are its secondary functions?
1.2 What are the other utilities that a compiler interacts with? Describe their functions.
1.3 What is a front end and back end of a compiler? What are the advantages of breaking up the compiler functionality into these two distinct stages?
1.4 What are the different phases in a compiler? Explain each one of them.
1.5 What is the difference between syntax analysis and semantic analysis? Give an example each for an error found by the compiler during syntax analysis and semantic analysis.
1.6 What is a 'pass' in a compiler? Differentiate between a multiple pass compiler and a single pass compiler.
1.7 Describe the common data structures used by a compiler.
1.8 Write a simple 'C' language 'Hello World' program and compile it with the 'gcc' compiler to generate an executable program. Invoke the 'gcc' compiler in verbose mode (–v) to identify all the utilities that are used during the compilation process.

SUMMARY

Each chapter material is accompanied by a summary section that gives the reader a quick glimpse of what has been learnt in the chapter broadly.

REVIEW QUESTIONS AND EXERCISES

5.1 A compiler can choose one of the two options (a) Translate the input source into intermediate code and then convert it to final machine code; (b) Directly generate the final machine code from the input source. What is the preferred option and why?
5.2 Describe the three address code form of the intermediate code. List out some of operators used in three address code with examples.
5.3 How can three address code be implemented in a compiler? Describe triples and indirect triples method of implementing TAC with examples.
5.4 Compare the different methods of implementing three address code.
5.5 How is an abstract syntax tree different from a parse tree? List out some of the nodes in the AST for a C compiler?
5.6 Translate a C statement '$a = b + c - (4*a*b + 3*c)$;' into TAC. How are the binary operators like +, –, etc., handled during the translation?
5.7 Translate an array reference statement '$a = b[c]$;' into TAC. What are the main TAC operators used during the translation? What attributes of a unary expression are used in translation of array references?
5.8 How is the offset calculated for a multidimensional array reference? Derive the formula.
5.9 Translate the C statements '*p*=& arr[3]; **p*=10;' into TAC. What TAC operators are useful during the translation of pointer accesses?
5.10 Translate the C statement '*x*.age = 30;' into TAC. Assume that the field 'age' is at an offset of 20 bytes from the base of the structure. What are the common TAC operators used during the translation of 'struct' references using the dot operator?
5.11 Translate the C statement 'ptr→age=20;'. Assume that the field 'age' is at an offset of 20 bytes from the base of the structure. What are the common TAC operators used during the translation of 'struct' references using the arrow operator?
5.12 Translate the C statement 'if ($a<b$){$x=y$;} m=20;' into TAC. In a single pass compiler, how is the translation of Boolean test expression ($a < b$) performed? How does it know about the labels to jump on being true or false?
5.13 Describe the backpatching technique. How is it used in the translation of an input C statement 'if ((a < b) || ($c < d$)) {$m = 20$;} else {$m = 10$;} $p = m$;'?
5.14 What are the data structures used during the translation of a 'while' statement? Illustrate the usage of those data structures during the translation of a C statement 'while ($i < b$){val = val *i; $i = i + 1$;} m = val;'?
5.15 How is a switch-case statement translated into TAC? Illustrate with an example.
5.16 What are the calling and returning sequences? List out the TAC instructions generated during both of these sequences by taking a sample C code snippet.
5.17 What is the sequence of events in the called function during a procedure call? Illustrate with an example.
5.18 How is a call to a procedure translated into TAC? Illustrate with an example.
5.19 State if the following statements are true or false:
(a) The separation of a compiler into front end and back end is helpful in retargeting of the compiler.
(b) The separation of a compiler into front end and back end helps in adding support for a new

REVIEW QUESTIONS AND EXERCISES

Review Questions and Exercises provided at the end of each chapter help the readers reinforce their learning.