Intermediate Code Generation

SZABIST – Department Of Computing

For course of Compiler Construction taught by Sir. Muhammad Shahzad in Fall 2020

Hamza Hussain – 1812264

Abdullah Shaikh – 1812255

Mustafa Ahmad Zaidi – 1812277

What is Intermediate Code?

- It is an intermediate representation of the source language, in a format which is very similar to assembly.
- We need an intermediate representation so don’t have to modify our frontend based on different machines. We can just write different backends for different machines. This process is called retargeting.
- It also enables us machine independent code optimization.

Intermediate Representations

- Graphical Representation (*Abstract Syntax Tree*)
- Postfix notation
- Three-Address code *(triples and quadruples):*

    result := arg1 op arg2

- Two-Address code

    result := op arg 1

Syntax Directed Translation

Production	Semantic Rule
$S \rightarrow \textbf{id} := E$	S.nptr := *mknode*(':=', *mkleaf*(**id**, **id**.entry), E.nptr)
$E \rightarrow E_1 + E_2$	E.nptr := *mknode*('+', E_1.nptr, E_2.nptr)
$E \rightarrow E_1 * E_2$	E.nptr := *mknode*('*', E_1.nptr, E_2.nptr)
$E \rightarrow - E_1$	E.nptr := *mknode*('uminus', E_1.nptr)
$E \rightarrow (E_1)$	E.nptr := E_1.nptr
$E \rightarrow \textbf{id}$	E.nptr := *mkleaf*(**id**, **id**.entry)

Three-Address Code

a := b * -c + b * -c

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

Linearized representation of a syntax tree

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

Linearized representation of a syntax DAG

Implementation of Three-Address Statements: Quads

#	*Op*	*Arg1*	*Arg2*	*Res*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Quads (quadruples)

Pro: easy to rearrange code for global optimization
Cons: lots of temporaries

Using Flex and Bison to emulate Three-Address Code

- Flex will be used to make a scan to recognize input in form of tokens.

- Bison/Yacc will be used to describe the semantic structure of our input, and generate entries for our table, Quadruple-Structure, which will later be printed on to console. These strings will emulate the structure of Three-Address Code.

lex.l:

```
[\t];
{NUMBER}+   { strcpy(yylval.str, yytext); return ID; }
{ALPHABET}  { strcpy(yylval.str, yytext); return ID; }
"while"                 { return WHILE; }
"do"        { return DO; }
"if"        { return IF;}
"<"         { yylval.symbol=yytext[0]; return OP; }
">"         { yylval.symbol=yytext[0]; return OP; }
"!="        { yylval.symbol=yytext[0]; return OP; }
"=="        { yylval.symbol=yytext[0]; return OP; }
[ \n\t] ;
.           { return yytext[0]; }
```

Yacc.y

```
w       :   WHILE { quadruple_entry_loop(); } '('con')'  DO block { quadruple_entry_do(); } ;

ifstmt  :   IF { ifstart(); } '(' con ')' { iftrue(); } block ;

con     :   ID OP ID { quadruple_entry($1,$2,$3); } ;

expr    :   expr '+' expr { quadruple_entry($1, '+', $3); strcpy($$,temp); }
        |   expr '-' expr { quadruple_entry($1, '-', $3); strcpy($$,temp); }
        |   expr '/' expr { quadruple_entry($1, '/', $3); strcpy($$,temp); }
        |   expr '*' expr { quadruple_entry($1, '*', $3); strcpy($$,temp); }
        |   '(' expr ')'  { strcpy($$,$2); }
        |   ID            { strcpy($$,$1); }
        ;
```

Complete specification files are separately attached

Sample Input (Assignment Statement)

a = b * 5 - 10

d = 2 / 10 + 50

c = (a + d)*100 - 30 *2

Output:

```
E:\5th semester\Compiler Construction\Project\test7>compiler.exe < sample.txt

t0 := b * 5
t1 := t0 - 10
   := a = t1
t2 := 2 / 10
t3 := t2 + 50
   := d = t3
t4 := a + d
t5 := t4 * 100
t6 := 30 * 2
t7 := t5 - t6
   := c = t7
E:\5th semester\Compiler Construction\Project\test7>
```

Sample Input (While Loops)

```
while(i<2) do
i = i + 1

while(j>5) do
j = j - 1

while(i!=j) do
i = (j-5)+1
```

Output:

```
E:\5th semester\Compiler Construction\Project\test7>compiler.exe < sample2.txt

t0 := i < 2
L0 := if t0
t1 := i + 1
   := i = t1
   := goto L0
   := else L1
t2 := j > 5
L1 := if t2
t3 := j - 1
   := j = t3
   := goto L1
   := else L2
t4 := i ! j
L2 := if t4
t5 := j - 5
t6 := t5 + 1
   := i = t6
   := goto L2
   := else L3
E:\5th semester\Compiler Construction\Project\test7>
```

Sample input (if-statement)

```
if( i < 5 )
        a = b * 50
        c = a + 100
if(j != 10)
        d = 4 * 100 / 60
        e = (d / 2)  *  40
```

Output:

```
E:\5th semester\Compiler Construction\Project\test7>compiler.exe < sample3.txt

t0 := i < 5
L0 := if t0
L0 := goto L1
L1 :=
t1 := b * 50
   := a = t1
t2 := a + 100
   := c = t2
t3 := j ! 10
L2 := if t3
L2 := goto L3
L3 :=
t4 := 4 * 100
t5 := t4 / 60
   := d = t5
t6 := d / 2
t7 := t6 * 40
   := e = t7
E:\5th semester\Compiler Construction\Project\test7>
```

Conclusion:

- Learned about using bison and writing non- ambiguous grammars.
- Learned about shift/reduce and reduce/reduce conflicts and how to deal with them.
- Learned the processes involved in intermediate code generation.
- Learned how to emulate a quadruple table and write intermediate code on console in C language.