CS31003: Compilers: Machine Independent Translation

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

August 14 2014

Three Address Code

- Concepts
  - Address
  - Instruction

  In general these could be classes, specializing for every specific type.
- Uses only up to 3 addresses in every instruction
- Every 3 address instruction is represented by a quad – opcode, argument 1, argument 2, and result

Three Address Code

- Address Types
  - *Name*:
    Source program names appear as addresses in 3-Address Codes.
  - *Constant*:
    Many different types and their (implicit) conversions are allowed as deemed addresses.
  - *Compiler-Generated Temporary*:
    Create a distinct name each time a temporary is needed - good for optimization.

Three Address Code

- Instruction Types
  For Addresses `x`, `y`, `z`, and Label `L`
  - *Binary Assignment Instruction*: For a binary `op` (including arithmetic, logical, or bit operators):

    `x = y op z`
  - *Unary Assignment Instruction*: For a unary operator `op` (including unary minus, logical negation, shift operators, conversion operators):

    `x = op y`
  - *Copy Assignment Instruction*:

    `x = y`

Three Address Code

- Instruction Types
  For Addresses `x`, `y`, and Label `L`
  - *Unconditional Jump*:

    `goto L`
  - *Conditional Jump*:
    - *Value-based*:

      `if x goto L`

      `if false x goto L`
    - *Comparison-based*: For a relational operator `op` (including $<$, $>$, $==$, $!=$, $\leq$, $\geq$):

      `if x relop y goto L`

Three Address Code

- Instruction Types
  For Addresses `p`, `x1`, `x2`, and `xN`
  - Procedure Call: A procedure call `p(x1, x2, ..., xN)` having `N` ≥ 0 parameters is coded as:

    ```
    param x1
    param x2
    ...
    param xN
    y = call p, N
    ```

    Note that `N` is not redundant as procedure calls can be nested.
  - Return Value: Returning a return value and /or assigning it is optional. If there is a return value it is returned from the procedure `p` as:

    ```
    return
    ```

Three Address Code

- Instruction Types
  For Addresses `x`, `y`, and `i`
  - *Indexed Copy Instructions*:

    ```
    x = y[i]
    x[i] = y
    ```

  - *Address and Pointer Assignment Instructions*:

    ```
    x = &y
    x = *y
    *x = y
    ```

Three Address Code

- Example

```
do i = i + 1; while (a[i] < v);
```

translates to

```
L: t1 = i + 1
   i = t1
   t2 = i * 8
   t3 = a[t2]
   if t3 < v goto L
```

The symbolic label is then given positional numbers as:

```
100: t1 = i + 1
101: i = t1
102: t2 = i * 8
103: t3 = a[t2]
100: if t3 < v goto 100
```

Three Address Code

- For

```
L: t1 = i + 1
   i = t1
   t2 = i * 8
   t3 = a[t2]
   if t3 < v goto L
```

quads are represented as:

	op	**arg 1**	**arg 2**	**result**
0	+	i	1	t1
1	=	t1		i
2	*	i	8	t2
3	=[]	a	t2	t3
4	<	t3	v	L

A Calculator Grammar

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

Yacc Specs (calc.y) for Calculator Grammar


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

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

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

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

%type <symp> expression
%%

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

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

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

Yacc Specs (calc.y) for Calculator Grammar


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

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

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

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

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

Note on Yacc Specs (calc.y)

- gentemp()
- emit()

Header (y.tab.h) for Calculator


```
/* A Bison parser, made by GNU Bison 2.5.  */
/* Tokens. */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers know about them. */
   enum yytokentype {
     NAME = 258,
     NUMBER = 259,
     UMINUS = 260
   };
#endif
/* Tokens. */
#define NAME 258
#define NUMBER 259
#define UMINUS 260

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

    int intval;
    struct symtab *symp;

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

extern YYSTYPE yylval;
```

Header (parser.h) for Calculator


```
#ifndef __PARSER_H
#define __PARSER_H

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

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

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

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

#endif // __PARSER_H
```

Flex Specs (calc.l) for Calculator Grammar


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

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

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

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


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

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

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

Sample Run

Output

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