Compilers (CS30003)

Lecture 18

Pralay Mitra

Autumn 2023-24

Intermediate Representation

- Components
  - Address
  - Instruction
- Restriction:
  - Up to three address (Three Address Code – TAC)
- Representations:
  - Quad (opcode, arg1, arg2, result)
  - Triples (opcode, arg1, arg2)
  - Indirect Triples (opcode, arg1, arg2)
  - Static Single Assignment (SSA)

Quadruples

A=B*-C+B*-C

	op	arg1	arg2	result	
0	minus	C		t1	t1=minus C
1	*	B	t1	t2	t2=B*t1
2	minus	C		t3	t3=minus C
3	*	B	t3	t4	t4=B*t3
4	+	t2	t4	t5	t5=t2+t4
5	=	t5		A	A=t5

Static Single Assignment (SSA) form

TAC	SSA
P=A+B	P1=A+B
Q=P-C	Q1=P1-C
P=Q*D	P2=Q1*D
P=E-P	P3=E-P2
Q=P+Q	Q2=P3+Q1

Φ- function

if(flag) x1= 1000; else x2=10;
x3=Φ(x1,x2);

Triples

A=B*-C+B*-C

	op	arg1	arg2	
0	minus	C		t1=minus C
1	*	B	t1	t2=B*t1
2	minus	C		t3=minus C
3	*	B	t3	t4=B*t3
4	+	t2	t4	t5=t2+t4
5	=	t5		A=t5

Indirect Triples

A=B*-C+B*-C

	op	arg1	arg2
0	minus	C	
1	*	B	t1
2	minus	C	
3	*	B	t3
4	+	t2	t4
5	=	t5	

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)
105	(5)

IR – address types

- Type of address
  - Name
    - Name of identifiers
  - Constant
    - int, float,….
    - Type casting (conversions) are allowed
  - Compiler generated temporary
    - Create variables required for TAC but not in source code
    - Generate distinct temporaries every time you need one – will be useful for optimization

IR – instruction types

- **Copy**
  x = y
- **Binary Assignment**
  x = y op z /* *op* may be arithmetic, logical or bitwise*/
- **Unary Assignment**
  x = op z /* *op* may be unary minus, logical negation, shift or type casting*/
- **Unconditional jump**
  goto L
- **Conditional jump**
  if x relop y goto L /* *relop* may be any relational operators: <,>,==,!=, ..*/
  ifFalse x relop y goto L /* *relop* may be any relational operators*/
- **Indexed copy**
  x = y[i] /* *i is an address* */
  y[i] = x /* *i is an address* */
- **Address and Pointer Assignment**
  x = &y
  x = *y
  ****y=x***

x, ***y***, ***z,*** and ***i*** are addresses; ***L*** is label

IR – instruction types

- **Function call**

  p(x1, x2)

  USAGE:

  ```
  int x1, x2
  x1=2
  x2=3
  xn=p(x1,x2)
  ```

  p, x1, x2, and ***xn*** are addresses; ***L*** is label

- **Return value**

  return x1 /* x1 may be optional */

Translation of Expression

- Operations within expressions

PRODUCTION	SEMNATIC RULE
S → id=E;	S.code=E.code || gen(top.get(id.lexval) '=' E.addr)
E → E1 + E2	E.addr=new Temp()
	E.code=E1.code||E2.code||gen(E.addr'='E1.addr '+' E2.addr)
E → - E1	E.addr=new Temp()
	E.code=E1.code||gen(E.addr'= 'minus' E2.addr)
E → (E1)	E.addr=E1.addr
	E.code=E1.code
E → id	E.addr=top.get(id.lexval)
	E.code=''

Translation of Expression

- Incremental Translation
- Addressing array elements

Translation of Expression

- Translation of array references

PRODUCTION	SEMANTIC ACTIONS
S → id=E;	{gen(top.get(id.lexval) '=' E.addr);}
S → L = E;	{gen(L.array.base '[' L.addr ']' '=' E.addr);}
E → E1 + E2	{E.addr=new Temp(); gen(E.addr '=' E1.addr '+' E2.addr);}
E → id	{E.addr=top.get(id.lexval);}
E → L	{E.addr=new Temp(); gen(E.addr '=' L.array.base '[' L.addr ']');}
L → id [E]	{L.array=top.get(id.lexval); L.type=L.array.type.elem; L.addr=new Temp(); gen(L.addr '=' E.addr '*' L.type.width);}

L → L1 [E]	{L.array=L1.array;
	L.type=L1.type.elem;
	t=new Temp();
	L.addr=new Temp();
	gen(t '=' E.addr '*' L.type.width);
	gen(L.addr '=' L1.addr '+' t);}

Homework

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

Translate to a three address code with positional numbers.

Handling Arithmetic Expression
A calculator grammar

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

Attributes and expression

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

Auxiliary method for translation

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

Expression grammar with Action

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