Course Code : CS 304

Course Name : Compiler Design

Prepared by : Vineetha K V

Assistant Professor, MEC

Reference : Aho A. Ravi Sethi and D Ullman. Compilers – Principles Techniques and Tools, Addison Wesley, 2006.

MODULE 5

Run-Time Environments:

Source Language issues, Storage organization, Storage-allocation strategies.

Intermediate Code Generation (ICG):

Intermediate languages – Graphical representations, Three-Address code, Quadruples, Triples. Assignment statements, Boolean expressions.

COURSE OUTCOME

At the end students may able to identify different storage allocation strategies and generate intermediate code for programming constructs.

PART II

INTERMEDIATE CODE GENERATION

Intermediate Code Generation

- *Intermediate codes* are machine independent codes, but they are close to machine instructions.
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- Intermediate language can be many different types, and the designer of the compiler decides this intermediate language.

Intermediate Languages Types

Graphical IRs:

- Abstract Syntax trees,
- DAGs,
- Control Flow Graphs

Linear IRs:

- Stack based (postfix)
- Three address code (quadruples)

Graphical IRs

Abstract Syntax Trees (AST)- retain essential structure of the parse tree, eliminating unwanted nodes.

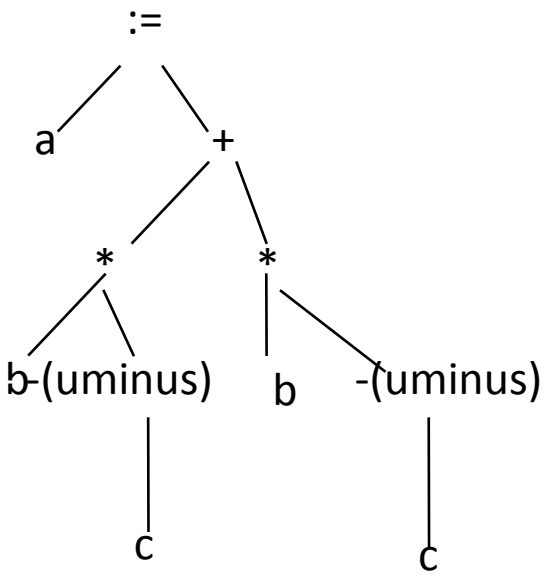
Directed Acyclic Graphs (DAG)- compacted AST to avoid duplication – smaller footprint as well

Control flow graphs (CFG)- explicitly model control flow

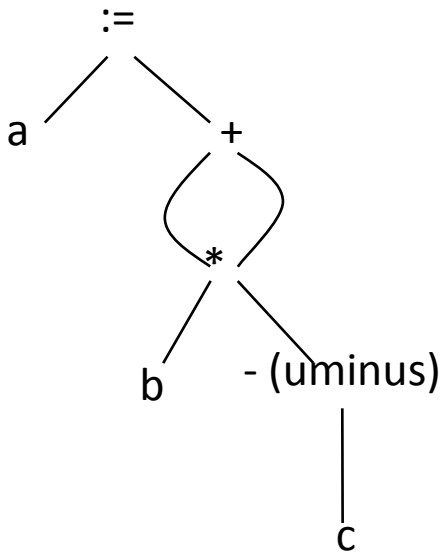
ASTs vs DAGs

- Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators.
- The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common sub expression would be replicated as many times as the sub expression appears in the original expression.
- DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

ASTs and DAGs:
a := b *-c + b*-c



AST



DAG

Syntax – directed definition to produce syntax tree or DAG

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**, **id**.$entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**, **num**.$val)$

Steps for constructing the DAG

1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Three-Address Code (Quadruples)

- Three address code (quadruple) is a sequence of instructions of the form:

  `x := y op z`

  where x, y and z are names, constants or compiler-generated temporaries; **op** stands for an operator.

- We may also use the following notation for quadruples (much better notation because it looks like a machine code instruction)

  `op y,z,x`

  apply operator `op` to `y` and `z`, and store the result in `x`.

- We use the term “three-address code” because each statement usually contains three addresses (two for operands, one for the result).

Three-Address Statements

1. ***Binary Operations:*** `op y,z,x` or `x := y op z`

   Assignment instructions of the form x = y op z, where op is a binary arithmetic or logical operation which is applied to `y` and `z`, and the result of the operation is stored in `x`. x, y, and z are addresses

   Ex:
   ```
   add  a,b,c
   gt   a,b,c
   ```

2. Unary Operations: `op y,,x` or `x := op y`

Assignments of the form x = op y, where `op` is a unary arithmetic or logical operator. This unary operator is applied to `y`, and the result of the operation is stored in x. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators

Ex:
```
uminus    a,,c
not       a,,c
inttoreal a,,c
```

Three-Address Statements (cont.)

3. Move Operations/ Copy instructions : `mov y,,x` or `x := y`

where the content of `y` is copied into x.

Ex:
```
mov   a,,c
movi  a,,c
movr  a,,c
```

4. Unconditional Jumps: `jmp ,,L` or `goto L`

We will jump to the three-address code with the label `L`, and the execution continues from that statement.

Ex:
```
jmp   ,,L1     // jump to L1
jmp   ,,7      // jump to the statement 7
```

Three-Address Statements (cont.)

5. Conditional Jumps: `jmp`***relop*** `y,z,L` or `if y` ***relop*** `z goto L`

We will jump to the three-address code with the label `L` if the result of `y relop z` is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement.

Ex:
```
jmpgt   y,z,L1  // jump to L1 if y>z
jmpgte  y,z,L1  // jump to L1 if y>=z
jmpe    y,z,L1  // jump to L1 if y==z
jmpne   y,z,L1  // jump to L1 if y!=z
```

Our relational operator can also be a unary operator.

```
jmpnz   y,,L1   // jump to L1 if y is not zero
jmpz    y,,L1   // jump to L1 if y is zero
jmpt    y,,L1   // jump to L1 if y is true
jmpf    y,,L1   // jump to L1 if y is false
```

Three-Address Statements (cont.)

6. Procedure Parameters: `param x,,` or `param x`

Procedure Calls: `call p,n,` or `y = call p,n`

where `x` is an actual parameter, we invoke the procedure `p` with `n` parameters.

Ex:

```
param x1,,
param x2,,
                ➔ p(x1,...,xn)
param xn,,
call  p,n,
```

```
f(x+1,y) ➔   add   x,1,t1
             param t1,,
             param y,,
             call  f,2,
```

Three-Address Statements (cont.)

7. Indexed Assignments:

```
move y[i],,x   or   x := y[i]
move x,,y[i]   or   y[i] := x
```

The instruction x = y[i] sets x to the value in the location i memory units beyond location y. The instruction x[i] = y sets the contents of the location i units beyond x to the value of y.

8. Address and Pointer Assignments:

```
moveaddr y,,x   or   x := &y
```

The instruction x =&Y sets the r-value of x to be the location (l-value) of y.

```
movecont y,,x   or   x := *y / *x := y
```

In the instruction x =*y, y is a pointer or a temporary whose r-value is a location. * x = y sets the r-value of the object pointed to by x to the r-value of y.

Implementation of Three- Address Statements

Implemented as records with fields for the operator and the operands.

Three representations :

- quadruples
- triples
- indirect triples

Quadruples

- A quadruple (or just "quad”) has four fields, op, arg1, arg2, and result.
- The op field contains an internal code for the operator. For instance, the three-address instruction x = y + z is represented by placing + in op, y in arg1, z in arg2, and x in result.
- The following are some exceptions to this rule:

  1. Instructions with unary operators like x = minus y or x = y do not use arg2.

  2. Operators like param use neither arg2 nor result.

  3. Conditional and unconditional jumps put the target label in result.

Triples

- A triple has only three fields, ie op, arg1, and arg2.
- There is no result field. It is referred by their position.

Indirect Triples

- Indirect triples consist of a listing of pointers to triples, rather than listing of triples themselves

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

Quadruple representation

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Assignment Statements

- In the syntax directed translation, assignment statement is mainly deals with expressions. The expression can be of type real, integer, array and records.
- Consider the grammar

$$S \rightarrow id := E$$
$$E \rightarrow E1 + E2$$
$$E \rightarrow E1 * E2$$
$$E \rightarrow (E1)$$
$$E \rightarrow id$$

Translation scheme to produce three address code for assignments

Production rule	**Semantic actions**

S → id :=E	{p = lookup(id.name);
	if p ≠ nil then
	emit (p = E.place)
	else error;
	}

E → E1 + E2	{E.place = newtemp(); emit (E.place = E1.place '+' E2.place) }
E → E1 * E2	{E.place = newtemp(); emit (E.place = E1.place '*' E2.place) }
E → (E1)	{E.place = E1.place}

E → id	{p = lookup(id.name);
	if p ≠ nil then
	E.place = p
	else error;
	}

Assignment Statements

- Translation scheme shows how symbol table entries can be found.
- The lexeme for the name represented by **id** is given by attribute *id.name*
- Operation *lookup(id.name)* checks if there is an entry for this occurrence of the name in the symbol table.
- If so, a pointer to the entry is returned; otherwise *lookup* returns *nil* to indicate that no entry was found
- The semantic actions use procedure *emit* to emit three address statements to an output file, rather than building up *code* attributes for nonterminals

Boolean Expressions

- Boolean expressions have two primary purposes. They are used for computing the logical values. They are also used as conditional expression using if-then-else or while-do.

  Consider the grammar

  E → E OR E
  E → E AND E
  E → NOT E
  E → (E)
  E → id relop id
  E → TRUE
  E → FALSE

  The relop is denoted by <, >, <, >.

  The AND and OR are left associated. NOT has the higher precedence then AND and lastly OR.

Boolean Expressions

Production rule	Semantic actions
E → E1 or E2	{E.place = newtemp(); emit (E.place ':=' E1.place ‘or' E2.place) }
E → E1 and E2	{E.place = newtemp(); emit (E.place ':=' E1.place ‘and' E2.place) }
E → not E1	{E.place = newtemp(); emit (E.place ':=' ‘not' E1.place) }
E → (E1)	{E.place = E1.place}

E → id relop id2	{E.place = newtemp();
	emit ('if' id1.place relop.op id2.place 'goto‘ nextstar + 3);
	emit (E.place ':=' '0');
	emit ('goto' nextstat + 2)
	emit (E.place ':=' '1')
	}

E → true	{E.place := newtemp(); emit (E.place ':=' '1') }
E → false	{E.place := newtemp(); emit (E.place ':=' '0') }

Boolean Expressions

- The *emit* function is used to generate the three address code and the *newtemp*( ) function is used to generate the temporary variables.
- The E → id relop id2 contains the next_state and it gives the index of next three address statements in the output sequence.

Translation of a<b or c<d and e<f

100: **if** a<b **goto** 103
101: t1:=0
102: **goto** 104
103: t1:=1
104: **if** c<d **goto** 107
105: t2:=0
106: **goto** 108
107: t2:=1
108: **if** e<f **goto** 111
109: t3:=0
110: **goto** 112
111: t3:= 1
112: t4:= t2 and t3
113: t5:= t1 or t4

Thank You