CSE 304 Compiler Design Intermediate Code Generation I

TONY MIONE

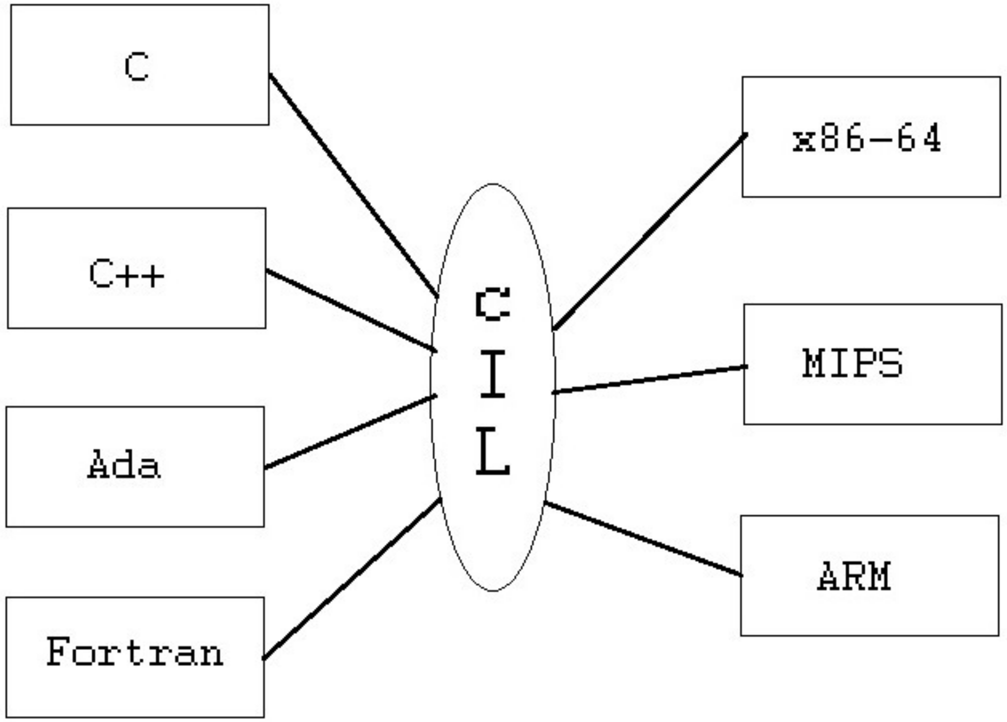
Topics

- Intermediate Code
- Intermediate representations
  - Trees
    - Syntax Trees
    - Directed Acyclic Graphs (DAGs)
    - Building DAGs
  - 3-address code
    - Quadruples
    - Triples
    - Indirect triples

Intermediate Code Generation

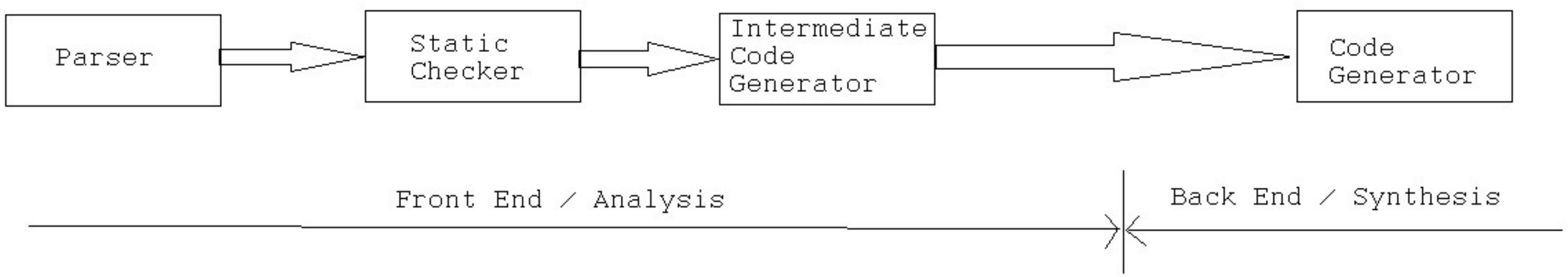
- Intermediate code is a 'bridge' between the analysis and synthesis phases of a compiler.
  - Well below the high level language structure
  - Still too abstract compared to target code
  - Can use it for machine independent optimization
- Good common intermediate code design can make development efficient:
  - Ex: To develop m different language compilers for n different architectures
    1. Must develop m * n separate compilers
    2. Or…
       - m language compiler front ends that produce common intermediate language
       - n target code generators

Intermediate Language



- Front ends generate same form of intermediate code
- Effectively, this is 12 compilers:
  - C -> x86-64
  - C -> MIPS
  - C -> ARM
  - C++ -> x86-64
  - C++ -> MIPS
  - C++ -> ARM
  - Ada -> x86-64
  - Ada -> MIPS
  - Ada -> ARM
  - Fortran -> x86-64
  - Fortran -> MIPS
  - Fortran -> ARM

Structure of Compiler Front End



- Above structure shows sequential operation of Parser->Static Checker -> Intermediate Code Generator
  - Can be merged into a single pass
  - Syntax Directed Definition / Translation Scheme can roll all these steps into the parser

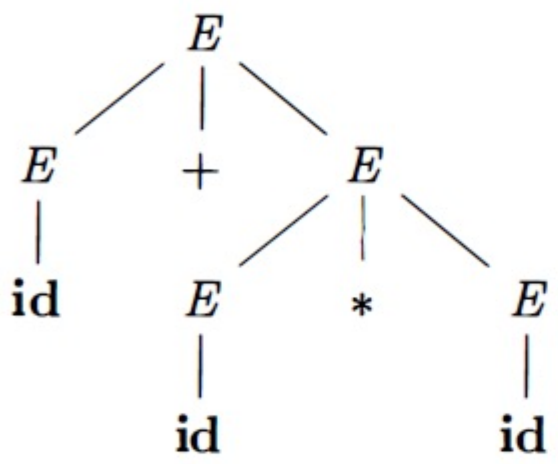
Intermediate Code

- Intermediate code can be designed at different 'levels' of representation
- Higher level intermediate code (like syntax trees)
  - show hierarchical structure of source code
  - Well suited for static type checking
- Lower level intermediate code (like quadruples)
  - Close to target architecture
  - Well suited for register allocation and instruction selection
- Compilers can use multiple intermediate forms

Source Program → High Level IR →… → Low level IR → Target Code

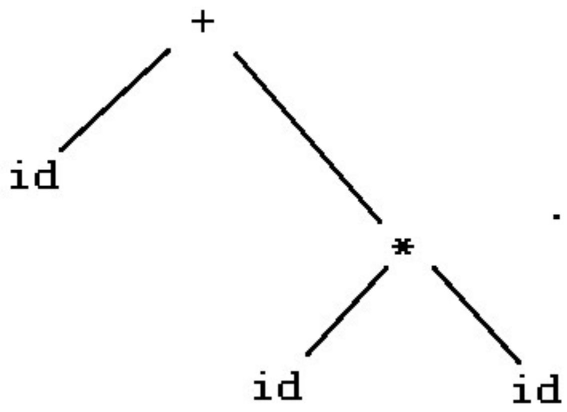
Trees

- Parse trees
  - Shows structure of code related to grammar
  - Not often (read: never) generated by a compiler
  - Too much grammar detail that does not help with code generation (includes terminals, non-terminals and even punctuation)



Trees

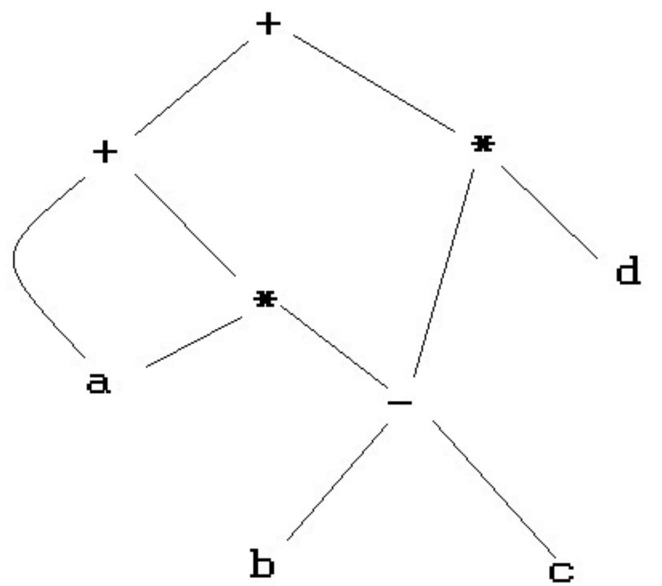
- Syntax Trees are much more terse and contain essential information
- Here is a tree equivalent to the parse tree on the last slide



Trees – Directed Acyclic Graphs

- Similar to Syntax Trees
- A node may have more than 1 parent
  - This identifies repeated uses of identifiers, values, and subexpressions
  - Helps generate more efficient code

Directed Acyclic Graph Example



DAG for:
a + a * (b – c) + (b – c) * d

Building DAGs

- The Value-Number method of for constructing DAGs
  - Typically, nodes of a syntax tree are kept in arrays of records
  - Each record holds:
    - opcode
    - Left and right children
  - Exception: Leaves have 1 additional node
    - A lexical value (number)
    - A pointer to a symbol table entry
- These records are in an array so each has an associated *index*.

Building DAGs – Value-Number Method

INPUT: Label op, node l , and node r .

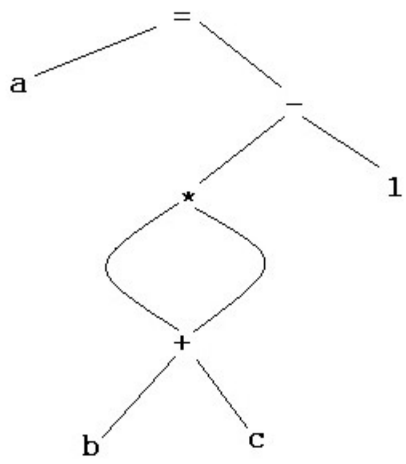
OUTPUT: The value number of a node in the array with signature (op, l, r) .

METHOD:

1. Search the array for a node M with label op, left child l, and right child r.
   a. If there is such a node , return the value number (index) of M.
   b. If not , create in the array a new node N with label op, left child l, and right child r, and return its value number (index).

Example: Building a DAG

a = (b + c) * (b + c) - 1



1	id	a	
2	id	b	
3	id	c	
4	+	2	3
5	*	4	4
6	num	1	
7	-	5	6
8	=	1	7
9			

3 Address Code

- Instructions usually contain 3 operands:
  - 2 source operands
  - 1 result operand
- There are a number of forms of 3-Address code
  - quadruples
  - triples
  - indirect triples

3 Address Code

- Types of operations available in a good intermediate form of 3 address code:
  - Assignment instructions (x = y op z)
  - Assignments with unary operators (x = op y)
  - Copy instructions (x = y)
  - Unconditional jumps (goto L)
  - Conditional Jumps (if x goto L, ifFalse x goto L)
  - Conditional Jumps with relationals (x relop y goto L)
    - relop is <, >, <=, >=, ==, !=
  - Procedure calls (param x, call p,n)
  - Indexed copy instructions (x = y[i], x[i] = y)
  - Address and pointer assignments (x=&y, x=*y, *x=y)

3 Address Code

- Choice of operations
  1. Operations in intermediate form must be rich enough to implement constructs of the source language
  2. Operations can be close to machine instructions instead
     a. Front end must generate long sequences of instructions for certain source constructs
     b. Makes work for optimizer and code generator more difficult to rediscover structure

Quadruples

- Have 4 fields:
  - Opcode *(op)*
  - 2 source operands *(arg1, arg2)*
  - 1 result *(result)*

- Some instructions do not use all 4 fields
  - Instructions with unary operators (x = minus y, x = y) do not use *arg2*
  - Operators like *param* do not use *arg2* or *result*
  - Conditional and unconditional jumps place the target label in *result*

Example: Quadruples

- Code for a = b * -c + b * -c;

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

(a) Three-address code

	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

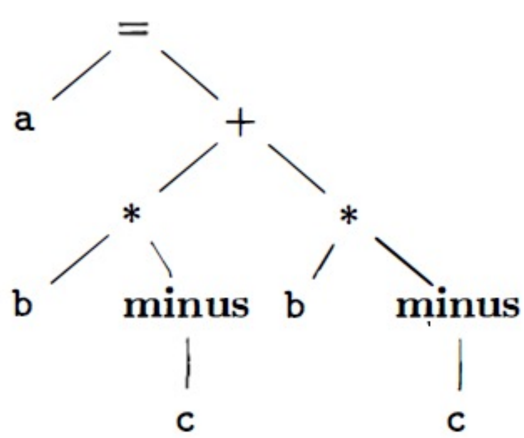
(b) Quadruples

Triples

- Triples have 3 fields:
  - opcode – an operation (*op)*
  - Two arguments – *arg1, arg2*
- Since the *result* field in quadruples is usually a temporary, triples just use the location of another triple as a source argument rather than writing to a temporary
- Triples produce problems for optimizers
  - Optimizers sometimes reorder instructions.
    - This is easy with quads since there is an explicit temporary variable.
    - With triples, results are based on the position in the instruction list meaning all references would have to be updated.

Triples

- Code for a = b * -c + b * -c;



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

(b) Triples

Indirect Triples

- These are like triples but add an extra array
- Instruction array holds a list of references to instructions in the triples array.
- An optimizer can reorder instructions by reordering the values in the instruction array and not touching the instructions in the triples structure.

Example: Indirect Triples

Code for a = b * -c + b * -c;

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)
	. . .

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	. . .		

Questions?