Compilers

by
Marwa Yusuf

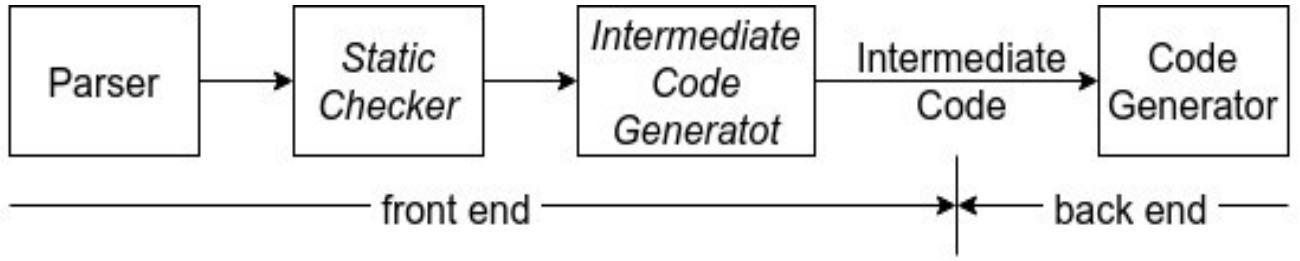
Lecture 9
Mon. 19-4-2021

Chapter 6 (6.1 to 6.2)

Intermediate Code Generation

Intro

- Front-end analyses source program and generates IR, from which back end generates machine code.
- With good IR, a compiler can be built by combining a front end for some language with the back end for some target machine. Using *m* front ends and *n* back ends, *m***n* compilers can be built.



Intro

- Static checking includes type checking, syntactic checks remaining after parsing (break statement enclosed within for, while or switch).
- IRs like syntax trees and three address code.

- Syntax-trees are high level, suitable for static type checking.
- Three address code can range depending on operators. (expressions vs loops).

Intro

- Choosing IR varies from compiler to another.
- IR may be actual language or just a set of data structures communicated between phases.
- C was used as IR for C++ original compiler.

Variants of Syntax Trees

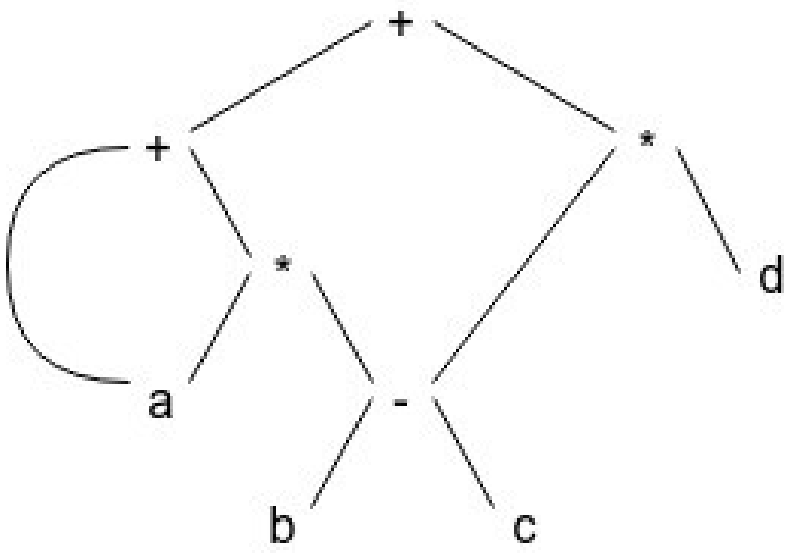
- Nodes (constructs), children (components).
- Directed Acyclic Graph (DAG): identifies the common sub-expressions of some expression.

DAG for Expressions

- The difference: a node can have more than one parent if it represents a common sub-expression (compared to repeating in syntax tree): hence more brief, more optimization.
- The same procedure to construct syntax tree, but with checking for existence before creation.

Example

Ex: a + a * (b – c) + (b – c) * d



	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*)

1) p_1 = *Leaf*(id, *entry-a*)

2) p_2 = *Leaf*(id, *entry-a*) = p_1

3) p_3 = *Leaf*(id, *entry-b*)

4) p_4 = *Leaf*(id, *entry-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*(id, *entry-b*) = p_3

9) p_9 = *Leaf*(id, *entry-c*) = p_4

10) p_{10} = *Node*('-', p_3 , p_4) = p_5

11) p_{11} = *Leaf*(id, *entry-d*)

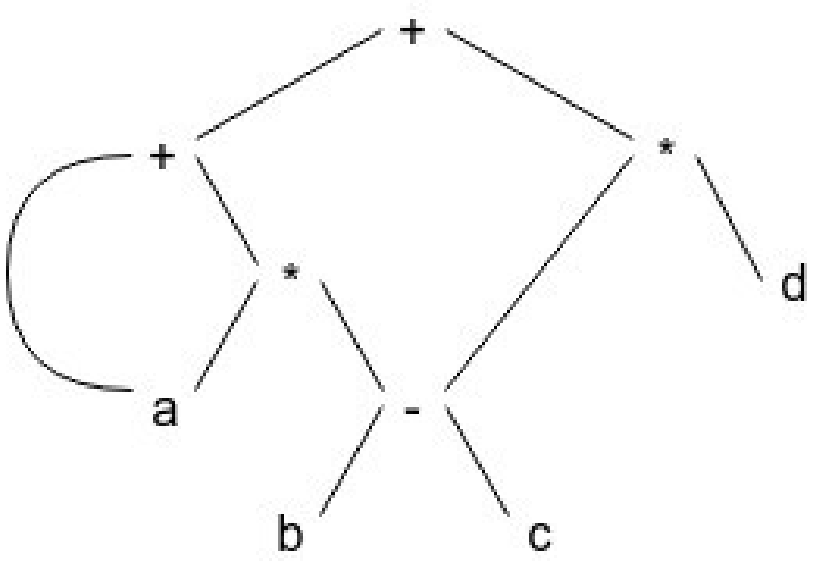
12) p_{12} = *Node*('*', p_5 , p_{11})

13) p_{13} = *Node*('+', p_7 , p_{12})

The Value-Number Method for Constructing DAG

- Skipped (6.1.2)

Three Address Code



$t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

- Linearized representation of a syntax tree or DAG where interior nodes correspond to explicit names.

Addresses and Instructions

- An address can be: a name (from source, in implementation: pointer to symbol table entry), a constant or a compiler-generated temporary.
- An instruction can be:
  - assignment *x = y op z*
  - assignment *x = op z* (unary operator like unary minus, negation, shift, conversion).
  - copy *x = y*
  - unconditional jump *goto L*
  - conditional jump *if x goto L* and *ifFalse x goto L*
  - conditional jump *if x relop y goto L*
  - procedure calls and returns *return y*

    param x1
    param x2
    ...
    param xn
    call p, n or *y = call p, n*
  - indexed copy *x = y[i]* and *x[i] = y*
  - address and pointer assignments *x = &y*, *x = *y* and **x = y*

Example

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

Symbolic Labels

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

Position Numbers

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

Three Address Code

- The choice of allowed operators:
  - More similar to machine, more easier to implement, but may produce long code, hence harder optimization and code generation afterwards.

Quadruples

- Need a representation of an instruction in a data structure.
- Quadruple (quad): op, arg_1, arg_2, result. Except:
  - Unary op: not use arg_2, in x = y → op is =, while in others it is implied.
  - Param: no arg2 nor result.
  - Jump: put target label in result.

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

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

	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
	...			

Triples

- Skipped (6.2.3)

Static Single Assignment Form

- SSA: IR that facilitates certain code optimizations.
- Differs from Three address Code in:

1) all assignments are to variables with distinct names (hence SSA name).

```
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
```

2) φ function

```
if (flag) x = -1; else x = 1;
y = x * a;
```

```
if (flag) x1 = -1; else x2 = 1;
x3 = Φ(x1, x2);
y = x3 * a;
```