Intermediate Representations

SLIDES SOURCED FROM THE DRAGON BOOK AND ENGINEERING A COMPILER

Introduction

A parser performs at least one pass over the source code to produce an internal representation of the source code.

It performs at least one pass over the internal representation to produce target code.

A compiler worth its name performs code optimization; a lot of it.

The internal representation used by the compiler to represent its knowledge of the source code is called an *intermediate representation* (IR).

Introduction

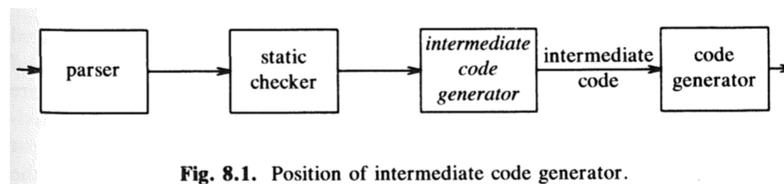


Fig. 8.1. Position of intermediate code generator.

Intermediate Representations

Three major categories:
- Structural
- Linear
- Hybrid

Structural Intermediate Representations

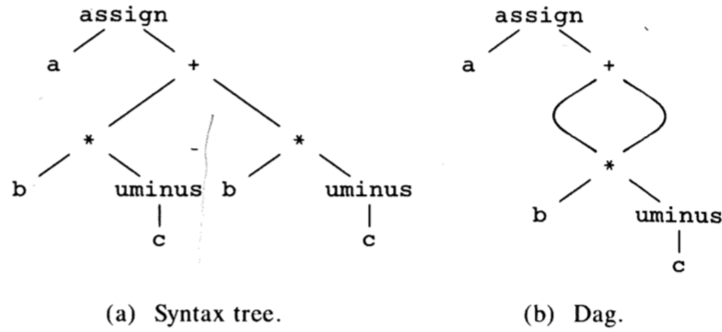
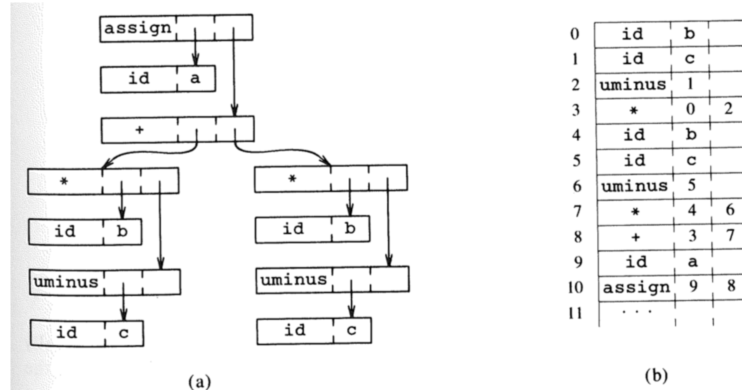


Fig. 8.2. Graphical representations of a := b*-c + b*-c.

Representation of Syntax Trees



0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	. . .		

Fig. 8.4. Two representations of the syntax tree in Fig. 8.2(a).

Generation of Syntax Trees

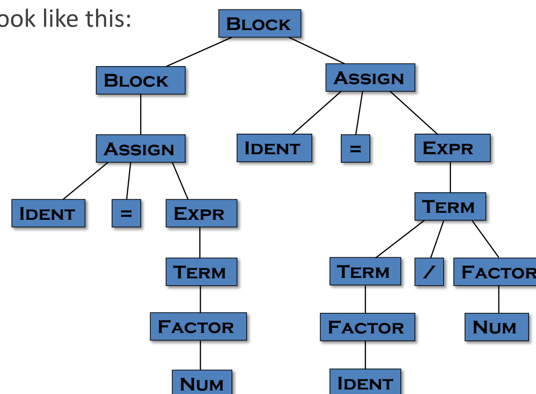
PRODUCTION	SEMANTIC RULE
$S \rightarrow$ **id** $:= E$	$S.nptr := mknode($'assign', $mkleaf($**id**, **id**$.place), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr := mknode($'+', $E_1.nptr, E_2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr := mknode($'*', $E_1.nptr, E_2.nptr)$
$E \rightarrow - E_1$	$E.nptr := mkunode($'uminus', $E_1.nptr)$
$E \rightarrow (E_1)$	$E.nptr := E_1.nptr$
$E \rightarrow$ **id**	$E.nptr := mkleaf($**id**, **id**$.place)$

Fig. 8.3. Syntax-directed definition to produce syntax trees for assignment statements.

Structural IRs: Parse Trees

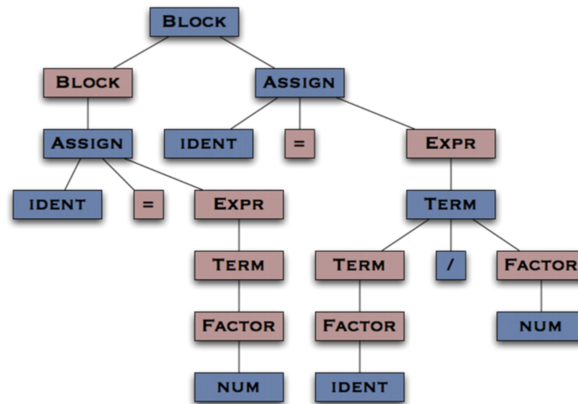
Consider the block: `{x = 42; y = x / 7.0;}`

The parse might look like this:



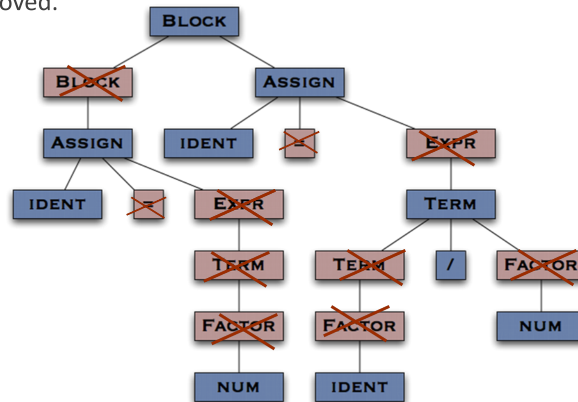
Structural IRs: Abstract Syntax Trees

A parse tree has extraneous nodes.



Structural IRs: Abstract Syntax Trees

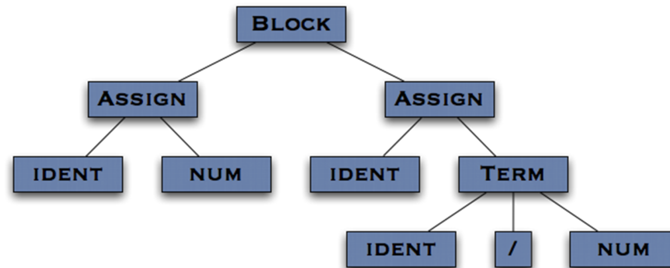
An Abstract Syntax Tree (AST), is like a parse tree with extraneous nodes removed.



Structural IRs: Abstract Syntax Trees

Resulting in the following structure for our block

```
{x = 42; y = x / 7.0;}
```



Linear IRs

Pseudo-assembly code for some abstract machine.

Many kinds have been used:

- **One address codes**. For processors that have a single accumulator register or for stack machines. For a while people built stack machines so as to execute Lisp code faster.
- **Two address codes**. Two argument registers, but result overwrote one of the registers. As such, they have destructive operations. The PDP-11 had such operations.
- **Three address codes**. For machines where most operations take two operands and produce a result. The rise of RISC architectures made these codes popular.

Stack-Machine Code

Operations performed by pushing operands on stack

Like an RPN calculator

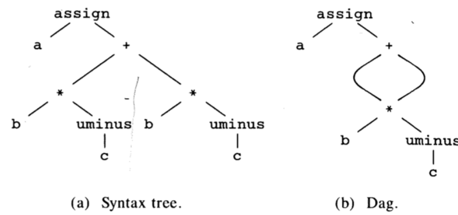
Used in Java Virtual Machine

Example: **a - 2 * b**

```
push 2
push b
multiply
push a
subtract
```

Three-Address Code

Most operations have the form **i ← j op k**
- two operands: j and k
- one operator: op
- one result: i



(a) Syntax tree. (b) Dag.

Fig. 8.2. Graphical representations of `a := b*-c + b*-c`.

(a) Code for the syntax tree.

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

(b) Code for the dag.

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

Fig. 8.5. Three-address code corresponding to the tree and dag in Fig. 8.2.

Hybrid Intermediate Representations

Combination of graphs and linear code
Example: control-flow graph

Structural IRs: Control Flow Graphs

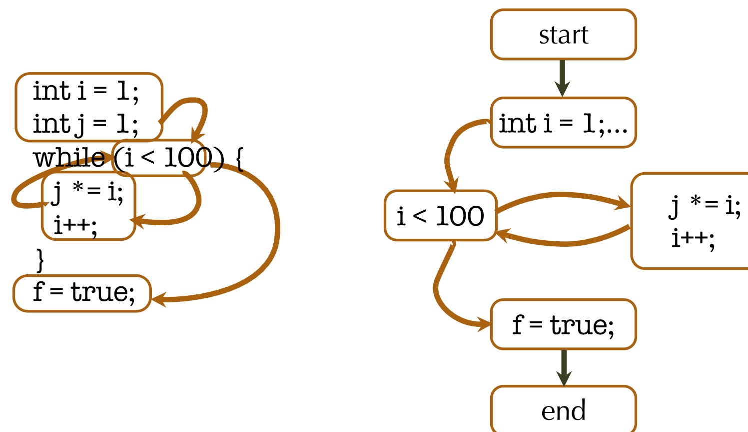
A *control flow graph* (CFG) is a graph *G* = (*N*, *E*) where

- Nodes n ∈ *N* correspond to basic blocks, i.e. sequences of operations that execute together.
- Directed edges e ∈ *E* correspond to transfer of control between blocks.

Used for:

- Optimization
- Instruction scheduling
- Register allocation

Example Control Flow Graph



Structural IRs: Dependence Graphs

A *dependence graph* is a graph *G* = (*N*, *E*) where

Nodes n ∈ *N* represent operations that

- Define data – "definition points"
- Use data – "use points"

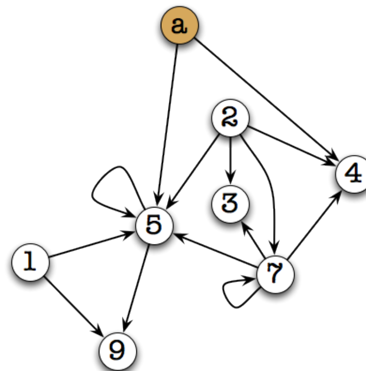
Directed edges e ∈ *E* from definition to use

Used for:

- Instruction scheduling
- Loop optimization

Example Dependence Graph

```
1        x = 0;
2        i = 1;
3        while (i < 100) {
4          if (a[i] > 0) {
5            x = x + a[i];
6          }
7          i = i + 1;
8        }
9        print(x);
```



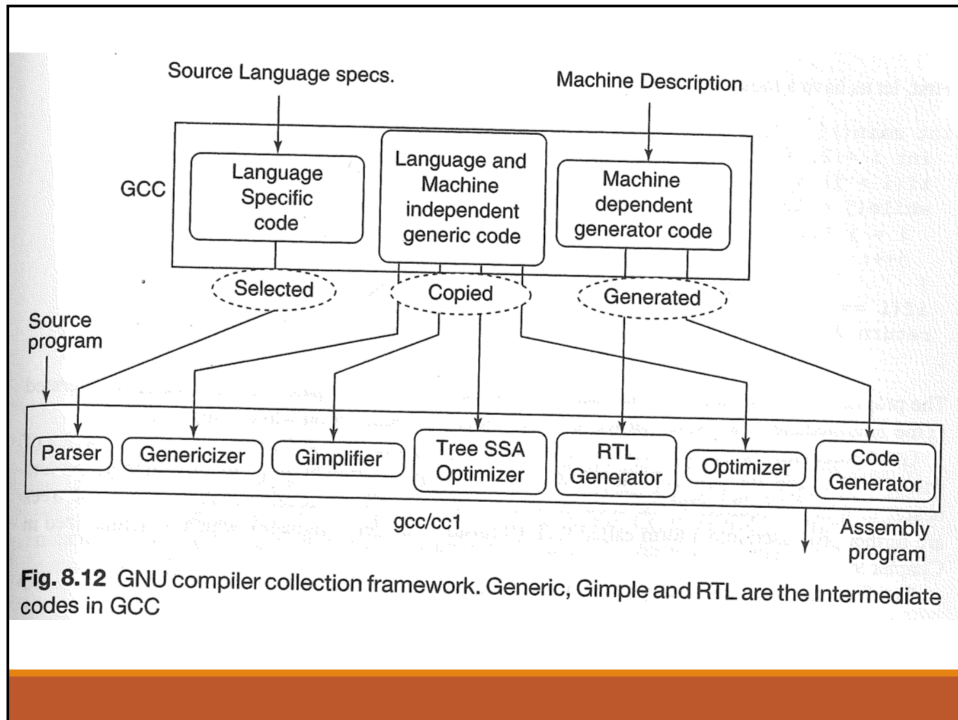


Fig. 8.12 GNU compiler collection framework. Generic, Gimple and RTL are the Intermediate codes in GCC