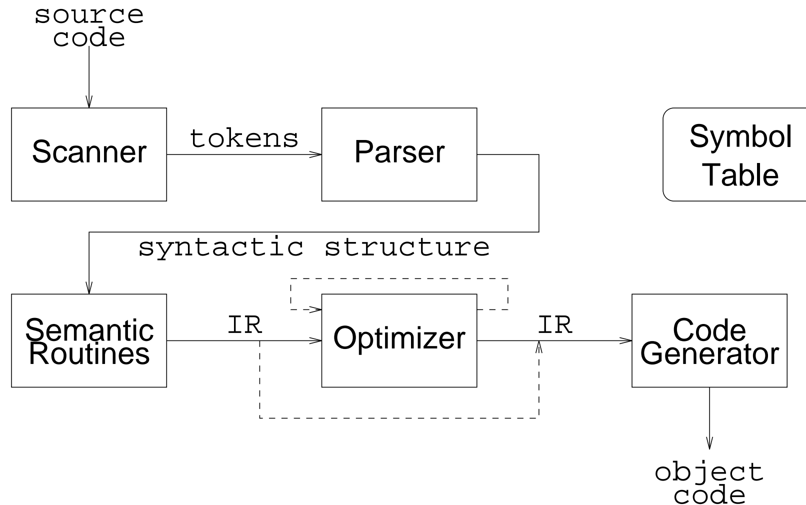
# Semantic Processing

Copyright ©2000 by Antony L. Hosking. *Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from hosking@cs.purdue.edu.*

# Alternatives for semantic processing




- one-pass analysis and synthesis
- one-pass compiler plus peephole
- one-pass analysis & IR synthesis + code generation pass
- multipass analysis (gcc)
- multipass synthesis (gcc)
- language-independent and retargetable compilers (gcc)

# One-pass compilers

- interleave scanning, parsing, checking, and translation
- no explicit IR
- generates target machine code directly
  emit short sequences of instructions at a time on each parser action (symbol match for predictive parsing/LR reduction)
  $\Rightarrow$ little or no optimization possible (minimal context)

Can add a *peephole optimization pass*

- extra pass over generated code through window (*peephole*) of a few instructions
- smoothes "rough edges" between segments of code emitted by one call to the code generator

# One-pass analysis/synthesis + code generation

*Generate explicit IR as interface to code generator*

- linear – e.g., tuples
- code generator alternatives:
  - one tuple at a time
  - many tuples at a time for more context and better code

Advantages

- back-end independent from front-end
  $\Rightarrow$ easier retargetting
  IR must be expressive enough for different machines
- add optimization pass later (multipass synthesis)

# Multipass analysis

Historical motivation: constrained address spaces

Several passes, each writing output to a file

1. scan source file, generate tokens (place identifiers and constants directly into symbol table)
2. parse token file
   generate *semantic actions* or linearized parse tree
3. parser output drives:
   - declaration processing to symbol table file
   - semantic checking with synthesis of code/linear IR

# Multipass analysis

Other reasons for multipass analysis (omitting file I/O)

- language may require it – e.g., declarations after use:
  1. scan, parse and build symbol table
  2. semantic checks and code/IR synthesis
- take advantage of tree-structured IR for less restrictive analysis: scanning, parsing, tree generation combined, one or more subsequent passes over the tree perform semantic analysis and synthesis

# Multipass synthesis

Passes operate on linear or tree-structured IR

Options

- code generation and peephole optimization
- multipass transformation of IR: machine-independent and machine-dependent optimizations
- high-level machine-independent IR to lower-level IR prior to code generation
- language-independent front ends (first translate to high-level IR)
- retargettable back ends (first transform into low-level IR)

# Multipass synthesis: e.g., GNU C compiler (`gcc`)

- language-dependent parser builds language-independent trees
- trees drive generation of machine-independent low-level **R**egister **T**ransfer **L**anguage for machine-independent optimization
- thence to target machine code and peephole optimization

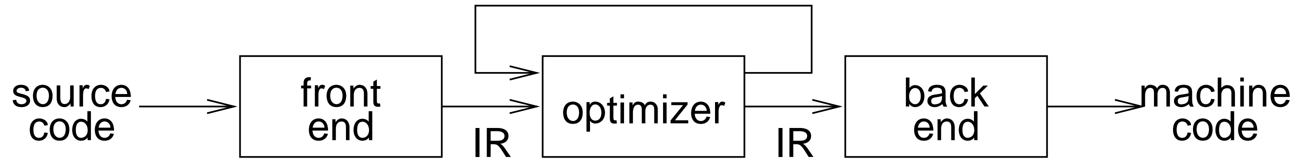
# Intermediate representations

*Why use an intermediate representation?*

1. break the compiler into manageable pieces
   good software engineering technique
2. allow a complete pass before code is emitted
   lets compiler consider more than one option
3. simplifies retargeting to new host
   isolates back end from front end
4. simplifies handling of “poly-architecture” problem
   $m$ lang’s, $n$ targets $\Rightarrow m+n$ components (*myth*)
5. enables machine-independent optimization
   general techniques, multiple passes

*An intermediate representation is a compile-time data structure*

# Intermediate representations




Generally speaking:

- front end produces IR
- optimizer transforms that representation into an equivalent program that may run more efficiently
- back end transforms IR into native code for the target machine

# Intermediate representations

*Representations talked about in the literature include:*

- abstract syntax trees (AST)
- linear (operator) form of tree
- directed acyclic graphs (DAG)
- control flow graphs
- program dependence graphs
- static single assignment form
- 3-address code
- hybrid combinations

# Intermediate representations

*Important IR Properties*

- ease of generation
- ease of manipulation
- cost of manipulation
- level of abstraction
- freedom of expression
- size of typical procedure
- original or derivative

Subtle design decisions in the IR have far reaching effects on the speed and effectiveness of the compiler.

Level of exposed detail is a crucial consideration.

# Intermediate representations

Broadly speaking, IRs fall into three categories:

Structural

- structural IRs are graphically oriented
- examples include trees, DAGs
- heavily used in source to source translators
- nodes, edges tend to be large

Linear

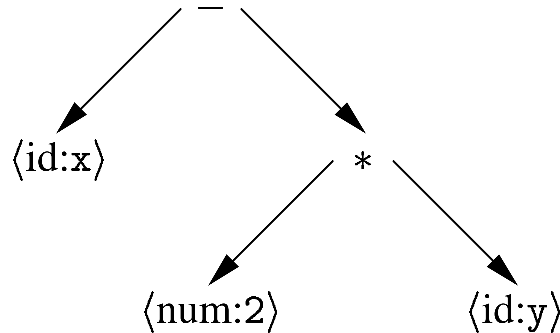
- pseudo-code for some abstract machine
- large variation in level of abstraction
- simple, compact data structures
- easier to rearrange

Hybrids

- combination of graphs and linear code
- attempt to take best of each
- e.g., control-flow graphs

# Abstract syntax tree

An abstract syntax tree (AST) is the procedure's parse tree with the nodes for most non-terminal symbols removed.




This represents "$\mathtt{x} - 2 * \mathtt{y}$".

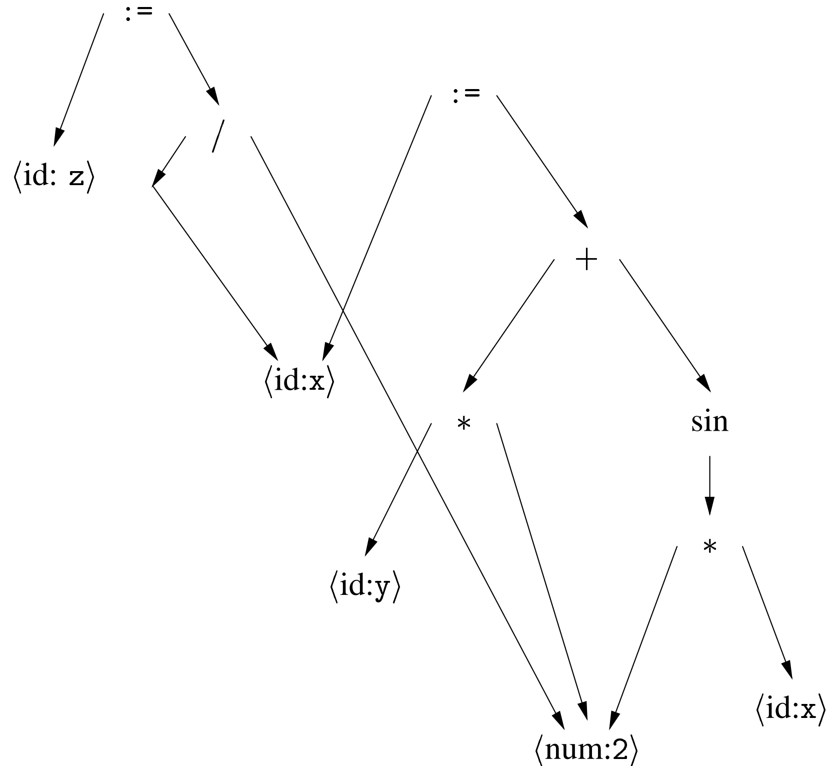
For ease of manipulation, can use a linearized (operator) form of the tree.

e.g., in postfix form: $\mathtt{x}\ 2\ \mathtt{y} * -$

# Directed acyclic graph

A directed acyclic graph (DAG) is an AST with a unique node for each value.

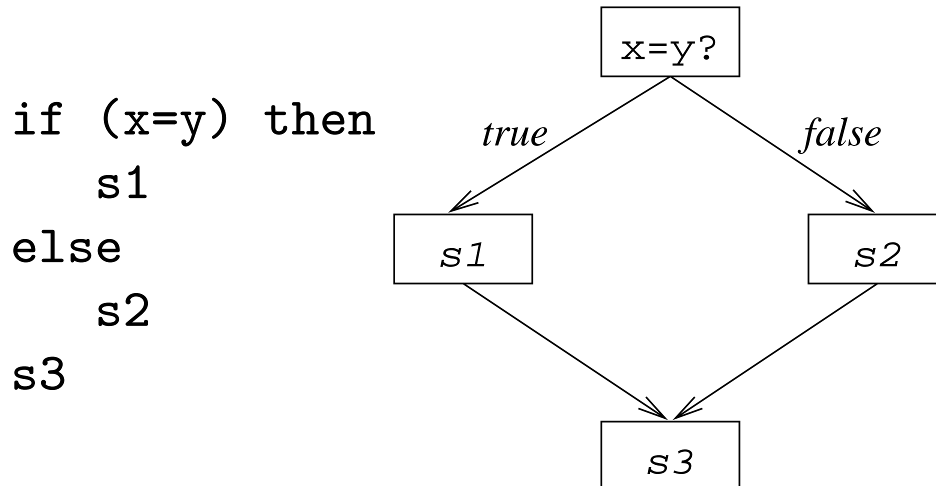
```
x := 2 * y + sin(2*x)
z := x / 2
```



# Control flow graph

The control flow graph (CFG) models the transfers of control in the procedure

- nodes in the graph are *basic blocks*
  straight-line blocks of code
- edges in the graph represent control flow
  loops, if-then-else, case, goto



# 3-address code

3-address code can mean a variety of representations.

In general, it allow statements of the form:

```
x ← y op z
```

with a single operator and, at most, three names.

Simpler form of expression:

```
x - 2 * y
```

becomes

```
t1 ← 2 * y
t2 ← x - t1
```

*Advantages*

- compact form (direct naming)
- names for intermediate values

Can include forms of prefix or postfix code

# 3-address code

Typical statement types include:

1. assignments
   `x ← y op z`
2. assignments
   `x ← op y`
3. assignments
   `x ← y[i]`
4. assignments
   `x ← y`
5. branches
   `goto L`
6. conditional branches
   `if x relop y goto L`
7. procedure calls
   `param x` and `call p`
8. address and pointer assignments

# 3-address code

Quadruples

```
x - 2 * y
```

| | | | | |
|---|---|---|---|---|
| (1) | load | t1 | y | |
| (2) | loadi | t2 | 2 | |
| (3) | mult | t3 | t2 | t1 |
| (4) | load | t4 | x | |
| (5) | sub | t5 | t4 | t3 |

- simple record structure with four fields
- easy to reorder
- explicit names

# 3-address code

Triples

```
x - 2 * y
```

| | | | |
|---|---|---|---|
| (1) | load | y | |
| (2) | loadi | 2 | |
| (3) | mult | (1) | (2) |
| (4) | load | x | |
| (5) | sub | (4) | (3) |

- use table index as implicit name
- require only three fields in record
- harder to reorder

# 3-address code

Indirect Triples

`x - 2 * y`

| | stmt | | op | arg1 | arg2 |
|---|---|---|---|---|---|
| (1) | (100) | (100) | load | y | |
| (2) | (101) | (101) | loadi | 2 | |
| (3) | (102) | (102) | mult | (100) | (101) |
| (4) | (103) | (103) | load | x | |
| (5) | (104) | (104) | sub | (103) | (102) |

- list of 1st triple in statement
- simplifies moving statements
- more space than triples
- implicit name space management

# Other hybrids

An attempt to get the best of both worlds.

- graphs where they work
- linear codes where it pays off

Unfortunately, there appears to be little agreement about where to use each kind of IR to best advantage.

For example:

- PCC and FORTRAN 77 directly emit assembly code for control flow, but build and pass around expression trees for expressions.
- Many people have tried using a control flow graph with low-level, three address code for each basic block.

# Intermediate representations

*But, this isn't the whole story*

Symbol table:

- identifiers, procedures
- size, type, location
- lexical nesting depth

Constant table:

- representation, type
- storage class, offset(s)

Storage map:

- storage layout
- overlap information
- (virtual) register assignments

# Advice

- Many kinds of IR are used in practice.
- There is no widespread agreement on this subject.
- A compiler may need several different IRs
- Choose IR with right level of detail
- Keep manipulation costs in mind

# Semantic actions

Parser must do more than accept/reject input; must also initiate translation.

*Semantic actions* are routines executed by parser for each syntactic symbol recognized.

Each symbol has associated *semantic value* (e.g., parse tree node).

Recursive descent parser:

- one routine for each non-terminal
- routine returns semantic value for the non-terminal
- store semantic values for RHS symbols in local variables

What about a table-driven LL(1) parser?

- maintain explicit *semantic stack* distinct from parse stack
- actions push results and pop arguments

# LL parsers and actions

How does an LL parser handle actions?

Expand productions *before* scanning `RHS` symbols, so:

- push actions onto parse stack like other grammar symbols
- pop and perform action when it comes to top of parse stack

# LL parsers and actions

```
push EOF
push Start Symbol
token ← next_token()
repeat
    pop X
    if X is a terminal or EOF then
        if X = token then
            token ← next_token()
        else error()
    else if X is an action
        perform X
    else /* X is a non-terminal */
        if M[X,token] = X → Y1Y2 ··· Yk then
            push Yk, Yk−1, ··· , Y1
        else error()
until X = EOF
```

# LR parsers and action symbols

What about LR parsers?

Scan entire RHS before applying production, so:

- cannot perform actions until entire RHS scanned
- can only place actions at very end of RHS of production
- introduce new marker non-terminals and corresponding productions to get around this restriction[†]

$$A \to w \textit{ action } \beta$$

becomes

$$A \to M\beta$$

$$M \to w \textit{ action}$$

[†]yacc, bison, CUP do this automatically