CS6013 - Modern Compilers: Theory and Practise

Semantic Analysis

V. Krishna Nandivada

IIT Madras

Semantic Processing

The compilation process is driven by the syntactic structure of the program as discovered by the parser

Semantic routines:

- interpret meaning of the program based on its syntactic structure
- two purposes:
  - finish analysis by deriving context-sensitive information
  - begin synthesis by generating the IR or target code
- associated with individual productions of a context free grammar or subtrees of a syntax tree

Context-sensitive analysis

What context-sensitive questions might the compiler ask?

1. Is `x` scalar, an array, or a function?
2. Is `x` declared before it is used?
3. Are any names declared but not used?
4. Which declaration of `x` does this reference?
5. Is an expression type-consistent?
6. Does the dimension of a reference match the declaration?
7. Where can `x` be stored? (heap, stack, . . .)
8. Does `*p` reference the result of a malloc()?
9. Is `x` defined before it is used?
10. Is an array reference in bounds?
11. Does function `foo` produce a constant value?
12. Can `p` be implemented as a memo-function?

These cannot be answered with a context-free grammar

Context-sensitive analysis

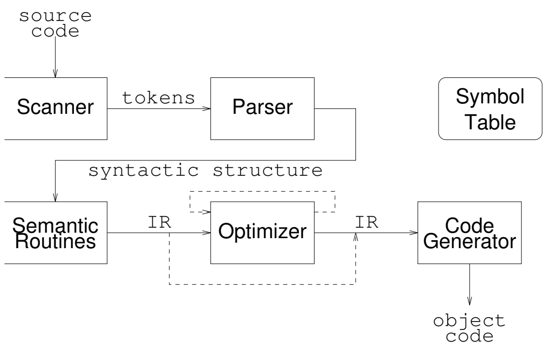
Why is context-sensitive analysis hard?

- answers depend on values, not syntax
- questions and answers involve non-local information
- answers may involve computation

Several alternatives:

abstract syntax tree (attribute grammars)	specify non-local computations automatic evaluators
symbol tables	central store for facts express checking code
language design	simplify language avoid problems

Alternatives for semantic processing



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

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)
  ⇒ 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
  ⇒ 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

Other reasons for multipass analysis (besides 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

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
- From RTL to target machine code and peephole optimization

Syntax directed translation

- 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).
- Semantic actions need to be specified for each production
- Challenges: How to execute the actions, how to specify the actions?

LL parsers and actions

How does an LL parser handle (aka - execute) 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] = $X \to Y_1 Y_2 \cdots Y_k$ then
            push $Y_k, Y_{k-1}, \cdots, Y_1$
        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

Action-controlled semantic stacks

Approach:

- stack is managed explicitly by action routines
- actions take arguments from top of stack
- actions place results back on stack

Advantages:

- actions can directly access entries in stack without popping (efficient)

Disadvantages:

- implementation is exposed
- action routines must include explicit code to manage stack (or use `stack` abstract data type).

LR parser-controlled semantic stacks

Idea: let parser manage the semantic stack
LR parser-controlled semantic stacks:

- parse stack contains already parsed symbols
- maintain semantic values in parallel with their symbols
- add space in parse stack or parallel stack for semantic values
- every matched grammar symbol has semantic value
- pop semantic values along with symbols

⇒ LR parsers have a very nice fit with semantic processing

LL parser-controlled semantic stacks

Problems:

- parse stack contains predicted symbols, not yet matched
- often need semantic value after its corresponding symbol is popped

Solution:

- use separate semantic stack
- push entries on semantic stack along with their symbols
- on completion of production, pop its RHS's semantic values

Specifying the actions: Attribute grammars

Idea: attribute the syntax tree

- can add attributes (fields) to each node
- specify equations to define values (unique)
- can use attributes from parent and children

Example: to ensure that constants are immutable:

- add type and class attributes to expression nodes
- rules for production on := that
  1. check that LHS.class is variable
  2. check that LHS.type and RHS.type are consistent or conform

Attribute grammars

To formalize such systems Knuth introduced attribute grammars:

- grammar-based specification of tree attributes
- value assignments associated with productions
- each attribute uniquely, locally defined
- label identical terms uniquely

Can specify context-sensitive actions with attribute grammars

Example

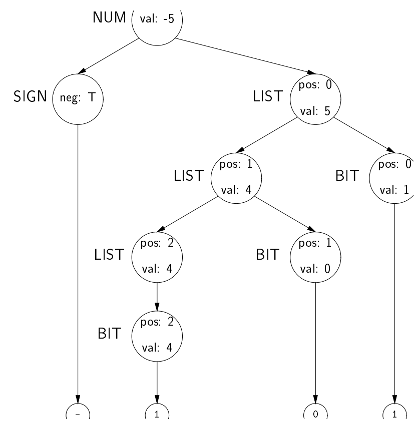
PRODUCTION	SEMANTIC RULES
$D \rightarrow T\ L$	$L.\text{in} := T.\text{type}$
$T \rightarrow$ **int**	$T.\text{type} := \text{integer}$
$T \rightarrow$ **real**	$T.\text{type} := \text{real}$
$L \rightarrow L_1$, **id**	$L_1.\text{in} := L.\text{in}$
	$\text{addtype}(\textbf{id}.\text{entry}, L.\text{in})$
$L \rightarrow$ **id**	$\text{addtype}(\textbf{id}.\text{entry}, L.\text{in})$

Example: Evaluate signed binary numbers

PRODUCTION	SEMANTIC RULES
NUM $\rightarrow$ SIGN LIST	LIST.pos := 0 if SIGN.neg NUM.val := -LIST.val else NUM.val := LIST.val
SIGN $\rightarrow$ +	SIGN.neg := false
SIGN $\rightarrow$ –	SIGN.neg := true
LIST $\rightarrow$ BIT	BIT.pos := LIST.pos LIST.val := BIT.val
LIST $\rightarrow$ LIST_1 BIT	LIST_1.pos := LIST.pos + 1 BIT.pos := LIST.pos LIST.val := LIST_1.val + BIT.val
BIT $\rightarrow$ 0	BIT.val := 0
BIT $\rightarrow$ 1	BIT.val := $2^{\text{BIT}.pos}$

Example (continued)

The attributed parse tree for `-101`:



- val and neg are synthetic attributes
- pos is an inherited attribute

Dependences between attributes

- values are computed from constants & other attributes
- synthetic attribute – value computed from children
- inherited attribute – value computed from siblings & parent
- key notion: induced dependency graph

The attribute dependency graph

- nodes represent attributes
- edges represent flow of values
- graph is specific to parse tree
- size is related to parse tree's size
- can be built alongside parse tree

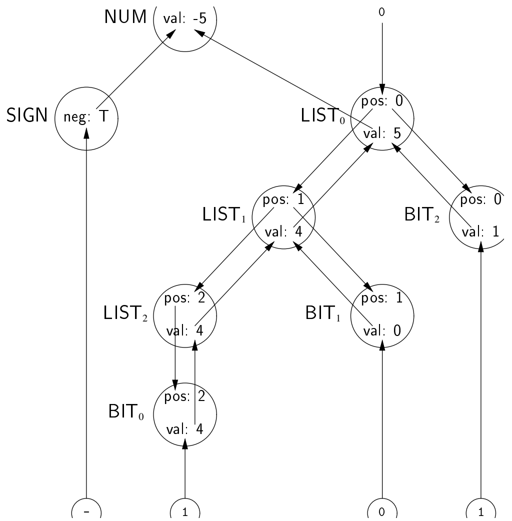
The dependency graph must be acyclic
Evaluation order:

- topological sort the dependency graph to order attributes
- using this order, evaluate the rules

The order depends on both the grammar and the input string

Example (continued)

The attribute dependency graph:



Example: A topological order

1. SIGN.neg
2. LIST_0.pos
3. LIST_1.pos
4. LIST_2.pos
5. BIT_0.pos
6. BIT_1.pos
7. BIT_2.pos
8. BIT_0.val
9. LIST_2.val
10. BIT_1.val
11. LIST_1.val
12. BIT_2.val
13. LIST_0.val
14. NUM.val

Evaluating in this order yields NUM.val: -5

Attribute Grammars

Advantages

- clean formalism
- automatic generation of evaluator
- high-level specification

Disadvantages

- evaluation strategy determines efficiency
- increased space requirements
- parse tree evaluators need dependency graph
- results distributed over tree
- circularity testing

Intel's 80286 Pascal compiler used an attribute grammar evaluator to perform context-sensitive analysis.
Historically, attribute grammar evaluators have been deemed too large and expensive for commercial-quality compilers.

Other uses

- the Cornell Program Synthesizer
- generate Ph.D. theses and papers
- odd forms of compiling — VHDL compiler
- structure editors for code, theorems, . . .

Attribute grammars are a powerful formalism

- relatively abstract
- automatic evaluation

Evaluation - for Type checking (MiniJava)

- We need generate type information.
  - For fields, variables, expressions, functions.
- Need to enforce types:
  - Assignments, function calls, expressions.
- We need to remember the type information and recall them as/where required.

Symbol tables

For compile-time efficiency, compilers use a symbol table:
- associates lexical names (symbols) with their attributes

What items should be entered?
- variable names
- defined constants
- procedure and function names
- literal constants and strings
- source text labels
- compiler-generated temporaries (we'll get there)

A symbol table is a compile-time structure
Separate table for structure layouts (types) (field offsets and lengths)

Symbol table information

What kind of information might the compiler need?
- textual name
- data type
- dimension information (for aggregates)
- declaring procedure
- lexical level of declaration
- storage class (base address)
- offset in storage
- if record, pointer to structure table
- if parameter, by-reference or by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions

Symbol table organization

How should the table be organized?
- Linear List
  - **O**(n) probes per lookup
  - easy to expand — no fixed size
  - one allocation per insertion
- Ordered Linear List
  - **O**$(\log_2 n)$ probes per lookup using binary search
  - insertion is expensive (to reorganize list)
- Binary Tree
  - **O**(n) probes per lookup — unbalanced
  - **O**$(\log_2 n)$ probes per lookup — balanced
  - easy to expand — no fixed size
  - one allocation per insertion
- Hash Table
  - **O**(1) probes per lookup — on average
  - expansion costs vary with specific scheme

Nested scopes: block-structured symbol tables

What information is needed?

- when asking about a name, want most recent declaration
- declaration may be from current scope or outer scope
- innermost scope overrides outer scope declarations

Key point: new declarations occur only in current scope

What operations do we need?

- `void put (Symbol key, Object value)`
  bind key to value
- `Object get(Symbol key)`
  return value bound to key
- `void beginScope()`
  remember current state of table
- `void endScope()`
  close current scope and restore table to state at most recent open beginScope

May need to preserve list of locals for the debugger

Nested scopes: complications

Fields and records:

give each record type its own symbol table

or assign record numbers to qualify field names in table

with R **do** ⟨stmt⟩:

- all IDs in ⟨stmt⟩ are treated first as R.id
- separate record tables:
  chain R's scope ahead of outer scopes
- record numbers:
  open new scope, copy entries with R's record number
  or chain record numbers: search using these first

Nested scopes: complications (cont.)

Implicit declarations:

- labels:
  declare and define name (in Pascal accessible only within enclosing scope)
- Ada/Modula-3/Tiger FOR loop:
  loop index has type of range specifier

Overloading:

- link alternatives (check no clashes), choose based on context

Forward references:

- bind symbol only after all possible definitions ⇒ multiple passes

Other complications:

packages, modules, interfaces — IMPORT, EXPORT

Attribute information

Attributes are internal representation of declarations
Symbol table associates names with attributes
Names may have different attributes depending on their meaning:

- variables: type, procedure level, frame offset
- types: type descriptor, data size/alignment
- constants: type, value
- procedures: formals (names/types), result type, block information (local decls.), frame size

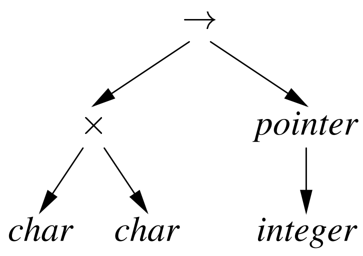
Type expressions

Type expressions are a textual representation for types:

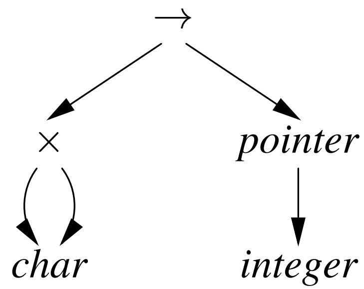
1. basic types: $boolean$, $char$, $integer$, $real$, etc.
2. type names
3. constructed types (constructors applied to type expressions):
   1. $array(I,T)$ denotes an array of T indexed over I
      e.g., $array(1\dots 10, integer)$
   2. products: $T_1 \times T_2$ denotes Cartesian product of type expressions T_1 and T_2
   3. records: fields have names
      e.g., $record((\mathtt{a} \times integer), (\mathtt{b} \times real))$
   4. pointers: $pointer(T)$ denotes the type "pointer to an object of type T"
   5. functions: $D \rightarrow R$ denotes the type of a function mapping domain type D to range type R
      e.g., $integer \times integer \rightarrow integer$

Type descriptors

Type descriptors are compile-time structures representing type expressions
e.g., $char \times char \rightarrow pointer(integer)$



or



Type compatibility

Type checking needs to determine type equivalence
Two approaches:

Name equivalence: each type name is a distinct type

Structural equivalence: two types are equivalent iff. they have the same structure (after substituting type expressions for type names)

- $s \equiv t$ iff. s and t are the same basic types
- $array(s_1, s_2) \equiv array(t_1, t_2)$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$
- $s_1 \times s_2 \equiv t_1 \times t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$
- $pointer(s) \equiv pointer(t)$ iff. $s \equiv t$
- $s_1 \rightarrow s_2 \equiv t_1 \rightarrow t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$

Type compatibility: example

Consider:

```
type  link  =  ↑cell;
var   next  :  link;
      last  :  link;
      p     :  ↑cell;
      q, r  :  ↑cell;
```

Under name equivalence:

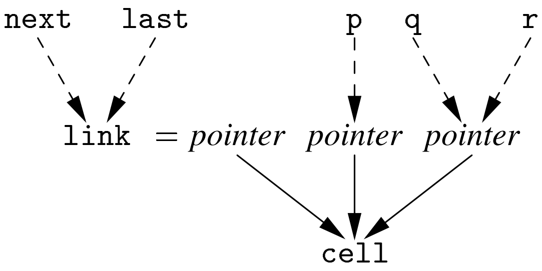
- `next` and `last` have the same type
- `p`, `q` and `r` have the same type
- `p` and `next` have different type

Under structural equivalence all variables have the same type
Ada/Pascal/Modula-2/Tiger are somewhat confusing: they treat distinct type definitions as distinct types, so `p` has different type from `q` and `r`

Type compatibility: Pascal name equivalence

Build compile-time structure called a type graph:

- each constructor or basic type creates a node
- each name creates a leaf (associated with the type's descriptor)



Type expressions are equivalent if they are represented by the same node in the graph

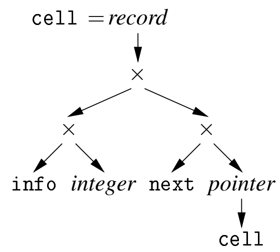
Type compatibility: recursive types

Consider:

```
type  link  =  ↑cell;
      cell  =  record
                 info :  integer;
                 next :  link;
               end;
```

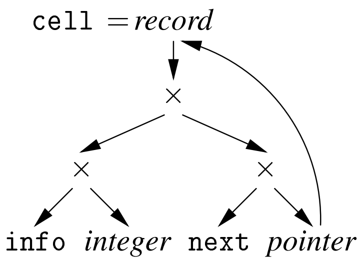
We may want to eliminate the names from the type graph

Eliminating name `link` from type graph for record:



Type compatibility: recursive types

Allowing cycles in the type graph eliminates `cell`:



Type checking minijava

- Populate symbol table

```
/**
 * f7 -> ( VarDeclaration() )*
 * f8 -> ( Statement() )*
 * f9 -> "return"
 * f10 -> Expression()
 */
public R visit(MethodDeclaration n, A argu) {
     currMethod = new currMethod(n, currClass);
      // build a symbol table for
      // the current method
     n.f7.accept();
     n.f8.accept();
     n.f10.accept();
}
```

Type checking minijava

- Populate symbol table(cont)

```
/**
 * f0 -> Type()
 * f1 -> Identifier()
 */
public R visit(VarDeclaration n, A argu) {
   R _ret=null;
   Type t = n.f0.accept();
   String id = n.f1.toString();
   if (currMethod == null) {
     if (!currClass.put(id, t))
       // error already defined in the class.
   } else if (!currMethod.put(id, t))
       // error already defined in the method.
   return _ret;
}
```


Type checking minijava

- Check for type correctness

```
/**
 * f0 -> PrimaryExpression()
 * f1 -> "+"
 * f2 -> PrimaryExpression()
 */
public R visit(PlusExpression n, A argu) {
   Type t1 = n.f0.accept(this, argu);
   if (! t1 instanceof IntegerType) ...
      // error -- lhs of plus expr
      //          should be integer
   Type t2 = n.f2.accept(this, argu);
   if (! t2 instanceof IntegerType)  ...
      // error -- rhs of plus expr
      //          should be integer
   return new IntegerType();
}
```


Food for though

- Overloaded addition operation.
- Assignment op.
- Function calls.
- Inheritance.

Storage classes of variables

During code generation, each variable is assigned an address (addressing method), approrpriate to its storage class.

- A local variable is not assigned a fixed machine address (or relative to the base of a module) – rather a stack location that is accessed by an offest from a register whose value does not point to the same location, each time the procedure is invoked. Why is it interesting?
- Four major storage classes: global, stack, stack static, registers

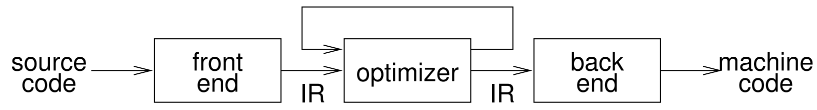
Intermediate representations

Why use an intermediate representation?

1. break the compiler into manageable pieces
   – good software engineering technique
2. simplifies retargeting to new host
   – isolates back end from front end
3. simplifies handling of “poly-architecture” problem
   – m lang’s, n targets $\Rightarrow m+n$ components (myth)
4. 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.

IR design issues

- Is the chosen IR appropriate for the (analysis/ optimization/ transformation) passes under consideration?
- What is the IR level: close to language/machine.
- Multiple IRs in a compiler: for example, High, Medium and Low

```
t1 ← a[i,j+2]        t1 ← j + 2          r1 ← [fp-4]
                     t2 ← i * 20         r2 ← r1 + 2
                     t3 ← t1 + t2        r3 ← [fp-8]
                     t4 ← 4 * t3         r4 ← r3 * 20
                     t5 ← addr a         r5 ← r4 + r2
                     t6 ← t5 + t4        r6 ← 4 * r5
                     t7 ← *t6            r7 ← fp - 216
                                         f1 ← [r7+r6]
(a)                  (b)                 (c)
```

(a) High-, (b) medium-, and (c) low-level representations of a C array reference.

- In reality, the variables etc are also only pointers to other data structures.

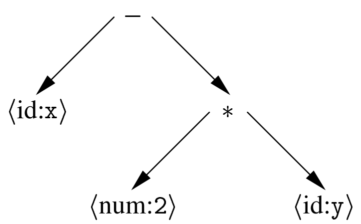
Intermediate representations

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
  - Example: GCC Tree IR.

Abstract syntax tree

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

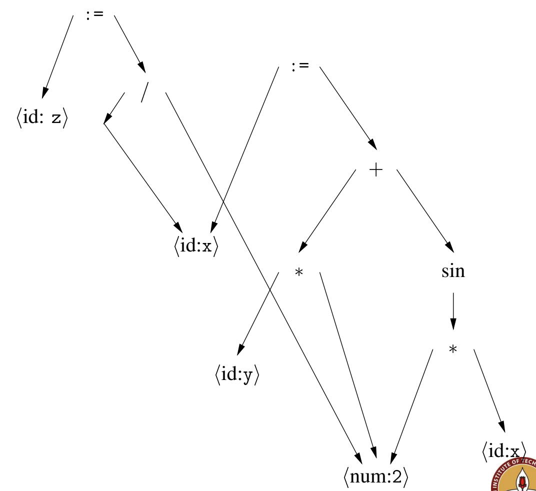


This represents "$x - 2 * y$".
For ease of manipulation, can use a linearized (operator) form of the tree.
e.g., in postfix form: $x\ 2\ y * -$

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
```



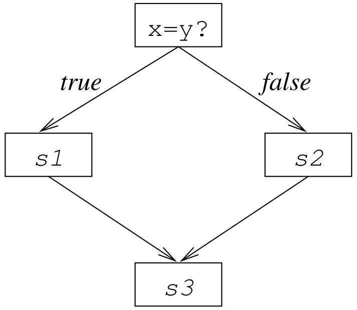
Q: What to do for matching names present across different functions?

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

```
if (x=y) then
   s1
else
   s2
s3
```



3-address code

- At most one operator on the right side of an instruction.
- 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: Addresses

Three-address code is built from two concepts: addresses and instructions.

- An address can be
  - A name: source variable program name or pointer to the Symbol Table same.
  - A constant: Constants in the program.
  - Compiler generated temporary:

3-address code

Typical instructions 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_1, `param` x_2, ... `param` x_n
   and
   `call p, n`
8. address and pointer assignments

How to translate:

`if (x) S1 else S2`

?

3-address code - implementation

Quadruples

- Has four fields: op, arg1, arg2 and result.
- Some instructions (e.g. unary minus) do not use arg2.
- For copy statement : the operator itself is =; for others it is implied.
- Instructions like `param` don't use neither arg2 nor result.
- Jumps put the target label in result.

`x - 2 * y`

op	arg1	arg2	result	
(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 - implementation

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 - implementation

Indirect Triples

`x - 2 * y`

	exec-order	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)

- simplifies moving statements (change the execution order)
- more space than triples
- implicit name space management

Indirect triples advantage

```
for i:=1 to 10 do
begin
 a=b*c
 d=i*3
end
     (a)
```

Optimized version

```
a=b*c
for i:=1 to 10 do
begin
 d=i*3
end
     (b)
```

```
(1)  := 1 i
(2)  * b c
(3)  := (2) a
(4)  * 3 i
(5)  := (4) d
(6)  + 1 i
(7)  LE I 10
(8)  IFT go (2)
```

Execution Order (a) : 12345678
Execution Order (b) : 23145678

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