Syntax-directed translation

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 functions:
  - 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

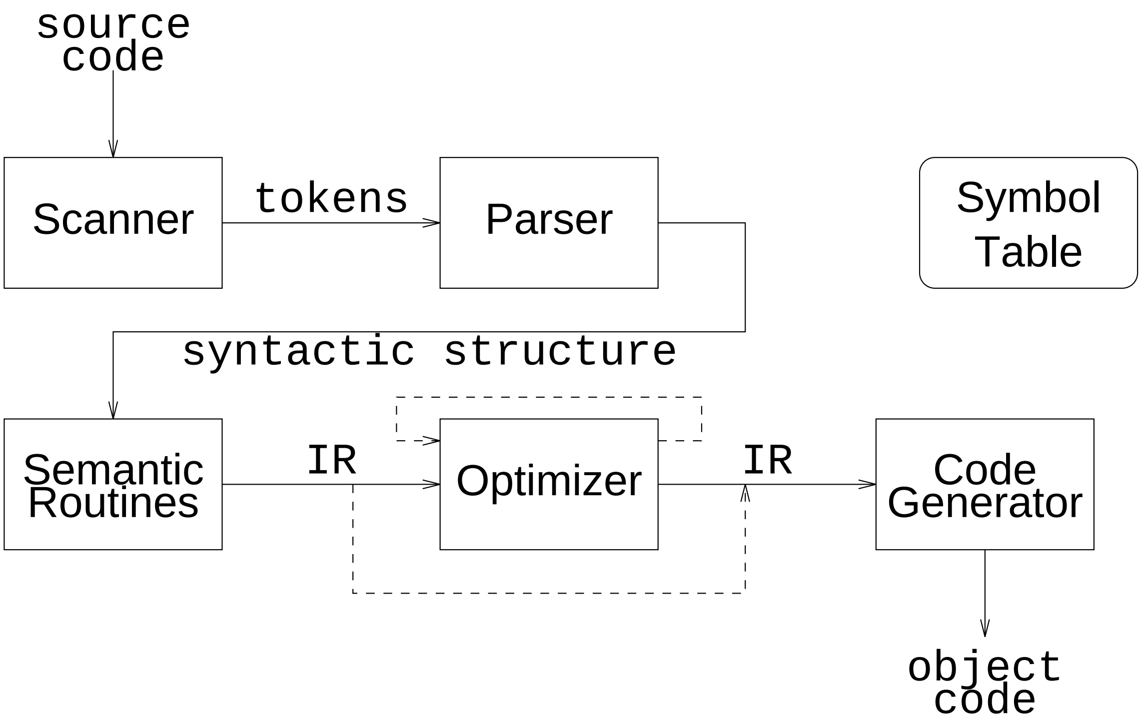
Why is context-sensitive analysis hard?

- 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

Alternative organizations for syntax-directed translation



- one-pass analysis and synthesis
- one-pass compiler plus peephole
- one-pass analysis and IR synthesis plus code generation pass
- multipass analysis
- multipass synthesis
- language-independent and retargetable 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/IR synthesis plus 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

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)

e.g., GNU C compiler (gcc):

- language-dependent parser builds language-independent trees
- trees drive generation of machine-independent low-level Register Transfer Language for machine-independent optimization
- thence to target machine code and peephole optimization

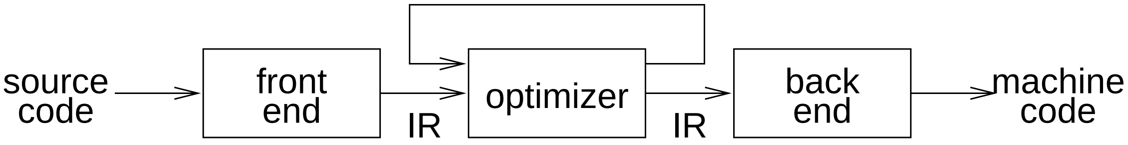
Intermediate representations

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.

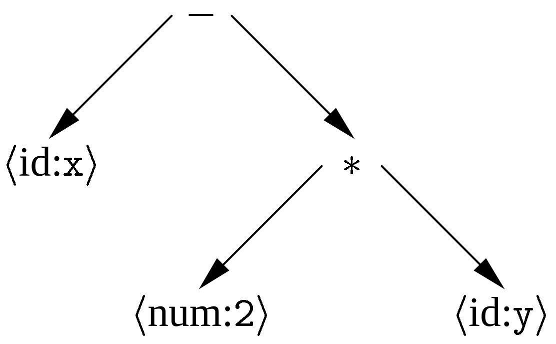
Intermediate representations

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
  - example would be CFG

Abstract syntax tree

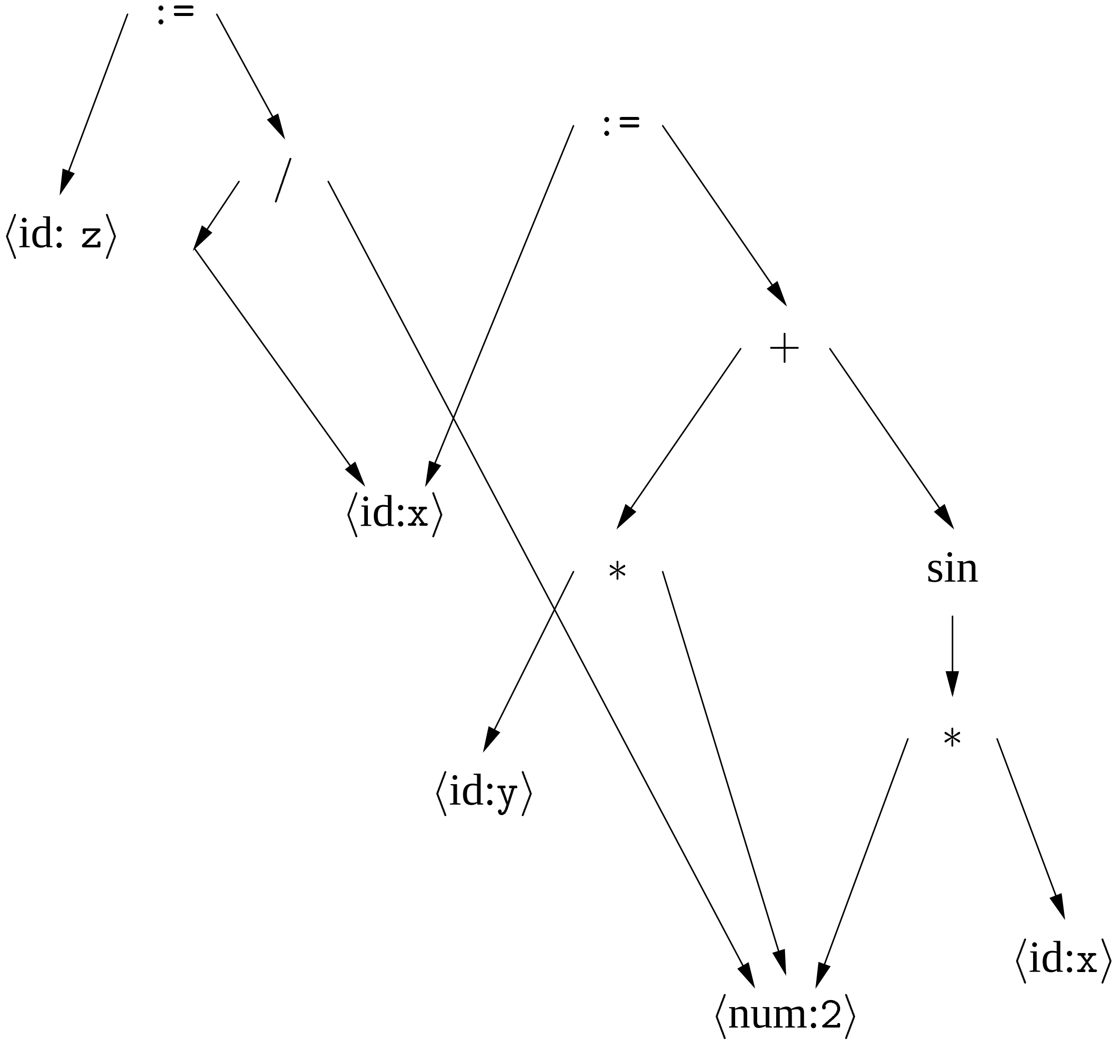
An abstract syntax tree (AST) is the procedure's parse tree with the nodes for most non-terminal symbols removed.



This represents "`x - 2 * y`".
For ease of manipulation, can use a linearised form of the tree.
`x 2 y * -` in postfix form

Directed acyclic graph

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



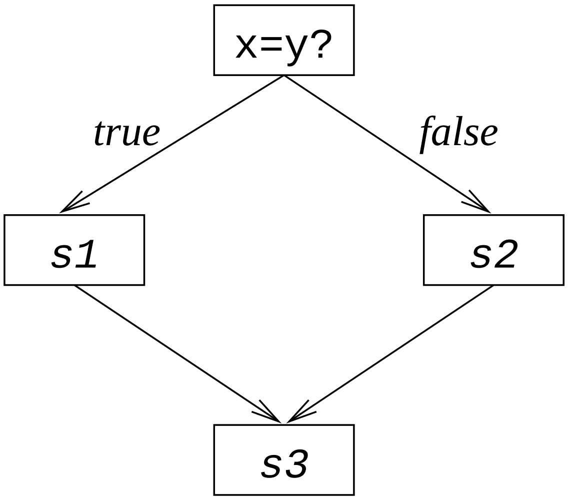
```
x := 2 * y + sin(2 * x)
z := x / 2
```

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

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



3-address code

3-address code is a term used to describe a variety of representations.
In general, they 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
(1)	(100)
(2)	(101)
(3)	(102)
(4)	(103)
(5)	(104)

	op	arg1	arg2
(100)	load	y	
(101)	loadi	2	
(102)	mult	(100)	(101)
(103)	load	x	
(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

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

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

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

- maintain explicit *semantic stack* separately 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:

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

```
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] == $X \rightarrow Y_1\ Y_2\ \ldots\ Y_k$ then
         push $Y_k$,$Y_{k-1}$,...,$Y_1$
      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

$$A \rightarrow w\ action\ \beta$$

becomes

$$\begin{gathered} A \rightarrow M\ \beta \\ M \rightarrow w\ action \end{gathered}$$

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

Alternative: *abstract semantic stacks*

- hide stack implementation behind push, pop interface
- accessing stack entries now requires pop (and copy to local var.)
- still need to manage stack within actions
  $\Rightarrow$ errors

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

LL parser-controlled semantic stacks: implementation

Keep 4 indices:

LHS	points to value for LHS symbol
RHS	points to value for first RHS symbol n^{th} RHS symbol is at RHS$+n-1$
current	points to RHS symbol being expanded
top	points to free entry on top of stack

Add new parse stack symbol EOP (end of production): contains indices to be restored upon completion of production

The modified skeleton LL parser

```
LHS = -1; RHS = -1; current = 0; top = 1
push EOF; push Start Symbol
token = next_token()
repeat
   pop X
   if X is a terminal or EOF then
      if X == token then
         sem_stack[current++] = token's semantic value
         token = next_token()
      else error()
   else if X is an action
      perform X
   else if X is EOP
      restore LHS, RHS, current, top
      current++ /* move to next symbol of prev. prod. */
   else /* X is a non-terminal */
      if M[X,token] == $X \rightarrow Y_1\ Y_2 ... Y_k$ then
         push EOP(LHS, RHS, current, top)
         push $Y_k, Y_{k-1}, ..., Y_1$
         LHS = current
         RHS = top
         top = top + k
         current = RHS
      else error()
until X == EOF
```

Attribute grammars

Idea: attribute the syntax tree

- can add attributes (*fields*) to each node
- specify equations to define values
- 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. checks that LHS.*class* is *variable*
  2. checks 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:

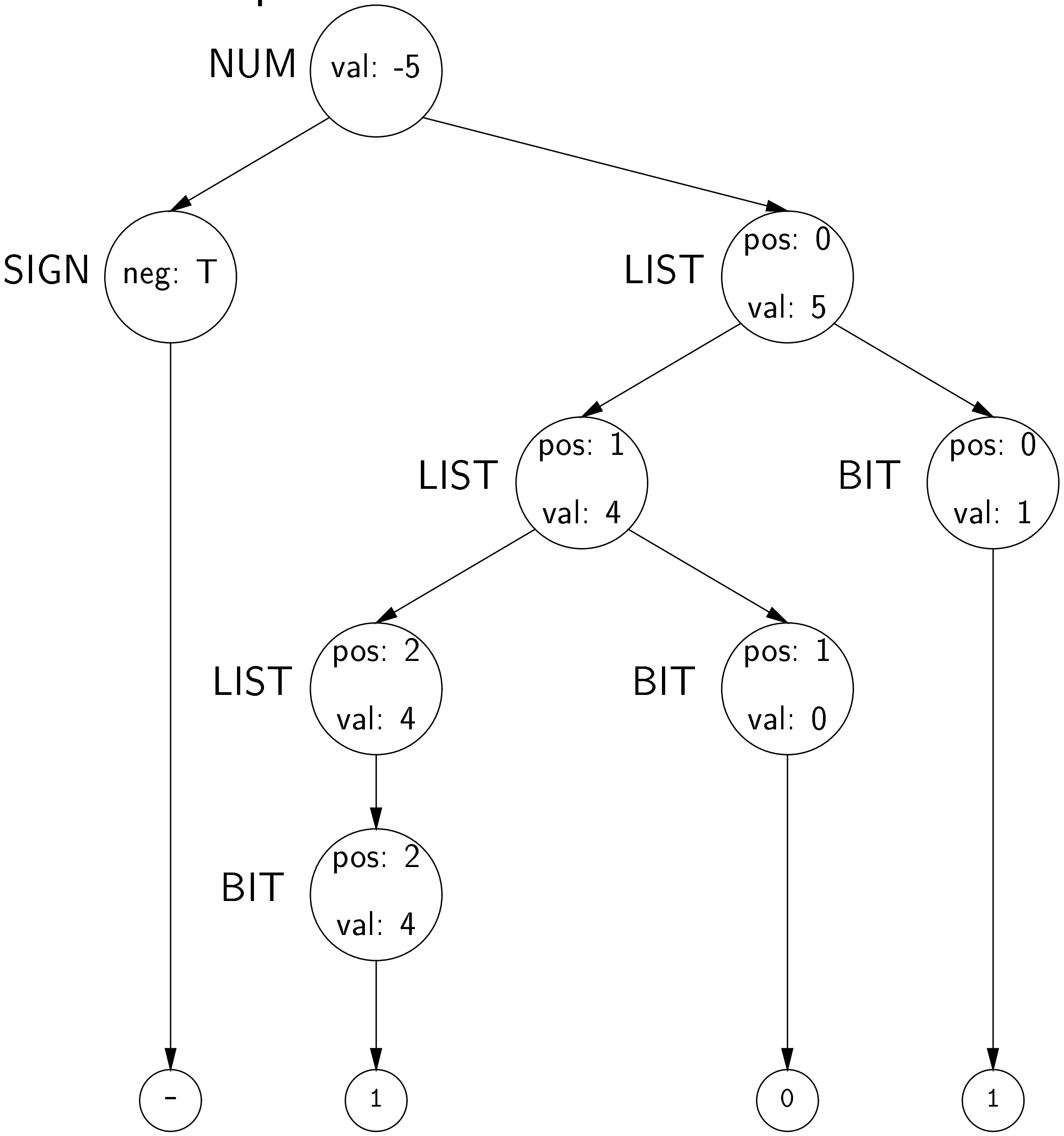
PRODUCTION	SEMANTIC RULES
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := integer
$T \rightarrow$ **real**	T.type := real
$L \rightarrow L_1$, **id**	L_1.in := L.in
	addtype(**id**.entry,L.in)
$L \rightarrow$ **id**	addtype(**id**.entry,L.in)

Example: A grammar to evaluate signed binary numbers

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

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

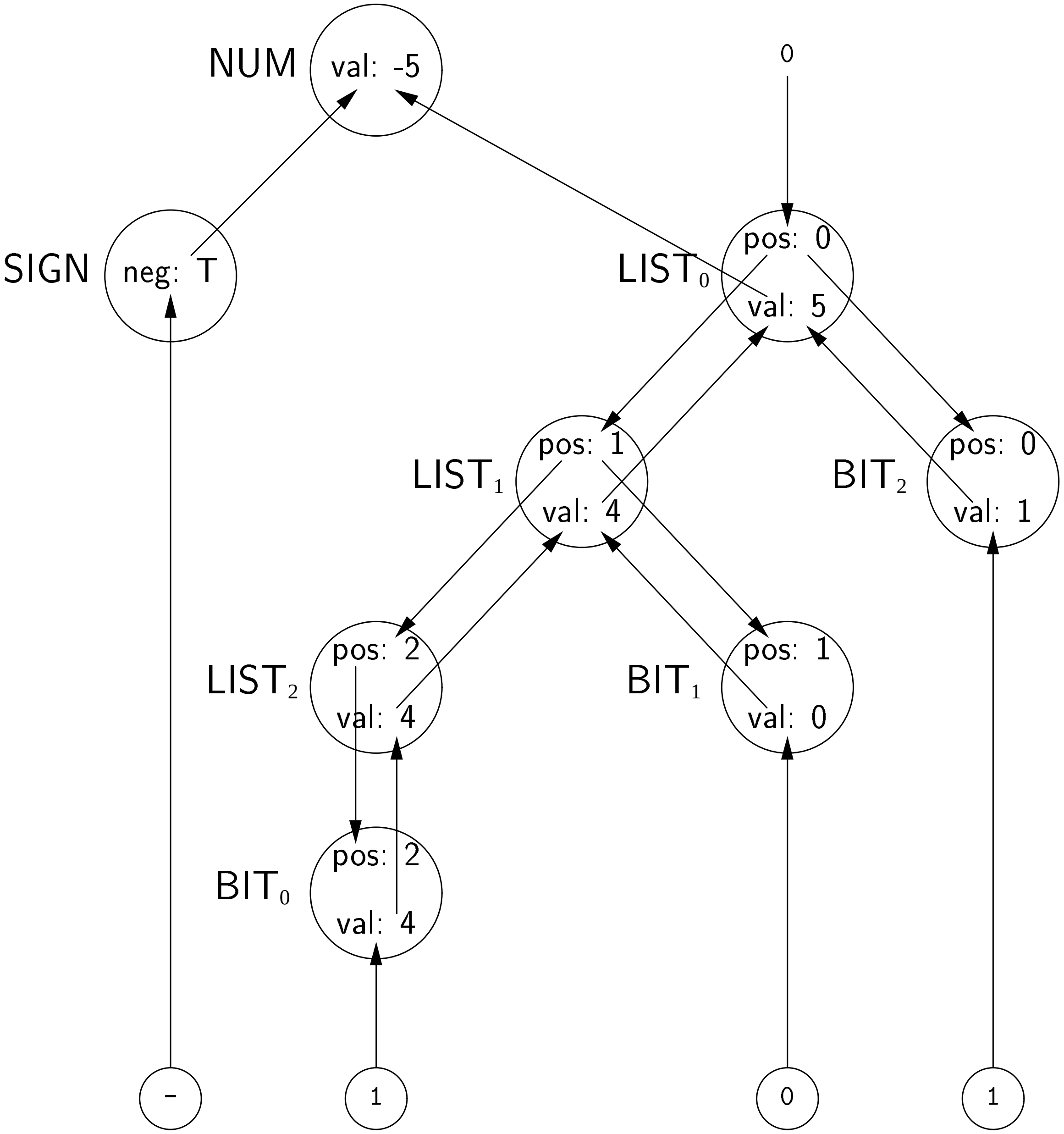
Evaluation order:

- 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

Evaluation strategies

Parse-tree methods (dynamic)

1. build the parse tree
2. build the dependency graph
3. topological sort the graph
4. evaluate it (cyclic graph fails)

Rule-based methods (treewalk)

1. analyse semantic rules at compiler-construction time
2. determine a static ordering for each production's attributes
3. evaluate its attributes in that order at compile time

Oblivious methods (passes)

1. ignore the parse tree and grammar
2. choose a convenient order (e.g., left-right traversal) and use it
3. repeat traversal until no more attribute values can be generated

Top-down (LL) on-the-fly one-pass evaluation

L-attributed grammar: given production $A \rightarrow X_1X_2 ... X_n$

- inherited attributes of X_j depend only on:
  1. inherited attributes of A
  2. arbitrary attributes of $X_1, X_2, ..., X_{j-1}$

- synthetic attributes of A depend only on its inherited attributes and arbitrary RHS attributes

- synthetic attributes of an action depends only on its inherited attributes

i.e., evaluation order:

Inh(A), Inh(X_1), Syn(X_1),..., Inh(X_n), Syn(X_n), Syn(A)

This is precisely the order of evaluation for an LL parser

Bottom-up (LR) on-the-fly one-pass evaluation

S-attributed grammar:

- L-attributed
- only synthetic attributes for non-terminals
- actions at far right of a RHS

Can evaluate S-attributed in one bottom-up (LR) pass

Inherited attributes: derive values from constants, parents, siblings

- used to express context (*context-sensitive* checking)
- inherited attributes are more “natural”

We want to use both kinds of attribute

- can always rewrite L-attributed LL grammars (using markers and copying) to avoid inherited attribute problems with LR

Bottom-up evaluation of inherited attributes

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

For copy rules generating inherited attributes value may be found at a fixed offset below top of stack

Simulating bottom-up evaluation of inherited attributes

Consider:

PRODUCTION	SEMANTIC RULES
$S \to aAC$	$C.i := A.s$
$S \to bABC$	$C.i := A.s$
$C \to c$	$C.s := g(C.i)$

C inherits synthetic attribute $A.s$ by copy rule

There may or may not be a B between A and C in parse stack

Rewrite as:

PRODUCTION	SEMANTIC RULES
$S \to aAC$	$C.i := A.s$
$S \to bABMC$	$M.i := A.s; C.i := M.s$
$C \to c$	$C.s := g(C.i)$
$M \to \epsilon$	$M.s := M.i$

Simulating bottom-up evaluation of inherited attributes

Consider:

PRODUCTION	SEMANTIC RULES
$S \rightarrow aAC$	$C.i := f(A.s)$

C inherits $f(A.s)$, but not by copying

Value of $f(A.s)$ is not in the stack

Rewrite as:

PRODUCTION	SEMANTIC RULES
$S \rightarrow aAMC$	$M.i := A.s; C.i := M.s$
$M \rightarrow \epsilon$	$M.s := f(M.i)$

Bottom-up (LR) on-the-fly one-pass evaluation

In general, an attribute grammar can be evaluated with one-pass LR if it is LC-attributed:

- L-attributed
- non-terminals in *left corners* have only synthetic attributes
- no actions in *left corners*

Left corners are that part of RHS sufficient to recognize the production, e.g., $A \rightarrow \alpha\beta$

LL(1) $\Rightarrow$ left corner α is empty

LR(1) $\Rightarrow$ left corner may be entire RHS (right corner β may be empty)

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.