

Review - LR(1) Example

Our simple grammar

1. $S' ::= S$
2. $S ::= L = R$
3. $S ::= R$
4. $L ::= *R$
5. $L ::= \text{id}$
6. $R ::= L$

Review - Canonical LR(1) collection

$$I_0 : \{ [S' ::= \bullet S, \text{eof}], [S ::= \bullet L = R, \text{eof}], \\ [S ::= \bullet R, \text{eof}], [L ::= \bullet * R, \{=, \text{eof}\}], \\ [L ::= \bullet \text{id}, \{=, \text{eof}\}], [R ::= \bullet L, \text{eof}] \}$$

$$I_1 : \{ [S' ::= S\bullet, \text{eof}] \}$$

$$I_2 : \{ [S ::= L\bullet = R, \text{eof}], [R ::= L\bullet, \text{eof}] \}$$

$$I_3 : \{ [S ::= R\bullet, \text{eof}] \}$$

$$I_4 : \{ [L ::= * \bullet R, \{=, \text{eof}\}], [R ::= \bullet L, \{=, \text{eof}\}], \\ [L ::= \bullet * R, \{=, \text{eof}\}], [L ::= \bullet \text{id}, \{=, \text{eof}\}] \}$$

$$I_5 : \{ [L ::= \text{id}\bullet, \{=, \text{eof}\}] \}$$

$$I_6 : \{ [S ::= L = \bullet R, \text{eof}], [R ::= \bullet L, \text{eof}], \\ [L ::= \bullet * R, \text{eof}], [L ::= \bullet \text{id}, \text{eof}] \}$$

$$I_7 : \{ [L ::= *R\bullet, \{=, \text{eof}\}] \}$$

$$I_8 : \{ [R ::= L\bullet, \{=, \text{eof}\}] \}$$

$$I_9 : \{ [S ::= L = R\bullet, \text{eof}] \}$$

$$I_{10} : \{ [R ::= L\bullet, \text{eof}] \}$$

$$I_{11} : \{ [L ::= * \bullet R, \text{eof}], [R ::= \bullet L, \text{eof}], \\ [L ::= \bullet * R, \text{eof}], [L ::= \bullet \text{id}, \text{eof}] \}$$

$$I_{12} : \{ [L ::= \text{id}\bullet, \text{eof}] \}$$

$$I_{13} : \{ [L ::= *R\bullet, \text{eof}] \}$$

LALR(1) parsing

LR(1) parsers have many more states than SLR(1) parsers (approximately factor of ten for Pascal).

LALR(1) parsers have same number of states as SLR(1) parsers, but with more power due to lookahead in states.

Define the *core* of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.

Thus, the two sets

- $\{[A ::= \alpha \bullet \beta, \mathbf{a}], [A ::= \alpha \bullet \beta, \mathbf{b}]\}$, and
- $\{[A ::= \alpha \bullet \beta, \mathbf{c}], [A ::= \alpha \bullet \beta, \mathbf{d}]\}$

have the same core.

Key Idea:

If two sets of LR(1) items, I_i and I_j , have the same core, we can merge the states that represent them in the ACTION and GOTO tables.

Back to our example grammar

- | | |
|------------------|----------------------|
| 1. $S' ::= S$ | 4. $L ::= *R$ |
| 2. $S ::= L = R$ | 5. $L ::= \text{id}$ |
| 3. $S ::= R$ | 6. $R ::= L$ |

$$I_0 : \{ [S' ::= \bullet S, \text{eof}], [S ::= \bullet L = R, \text{eof}], \\ [S ::= \bullet R, \text{eof}], [L ::= \bullet * R, \{=, \text{eof}\}], \\ [L ::= \bullet \text{id}, \{=, \text{eof}\}], [R ::= \bullet L, \text{eof}] \}$$

$$I_1 : \{ [S' ::= S \bullet, \text{eof}] \}$$

$$I_2 : \{ [S ::= L \bullet = R, \text{eof}], [R ::= L \bullet, \text{eof}] \}$$

$$I_3 : \{ [S ::= R \bullet, \text{eof}] \}$$

$$I_{4/11} : \{ [L ::= * \bullet R, \{=, \text{eof}\}], [R ::= \bullet L, \{=, \text{eof}\}], \\ [L ::= \bullet * R, \{=, \text{eof}\}], [L ::= \bullet \text{id}, \{=, \text{eof}\}] \}$$

$$I_{5/12} : \{ [L ::= \text{id} \bullet, \{=, \text{eof}\}] \}$$

$$I_6 : \{ [S ::= L = \bullet R, \text{eof}], [R ::= \bullet L, \text{eof}], \\ [L ::= \bullet * R, \text{eof}], [L ::= \bullet \text{id}, \text{eof}] \}$$

$$I_{7/13} : \{ [L ::= *R \bullet, \{=, \text{eof}\}] \}$$

$$I_{8/10} : \{ [R ::= L \bullet, \{=, \text{eof}\}] \}$$

$$I_9 : \{ [S ::= L = R \bullet, \text{eof}] \}$$

Example

- | | |
|------------------|----------------------|
| 1. $S' ::= S$ | 4. $L ::= *R$ |
| 2. $S ::= L = R$ | 5. $L ::= \text{id}$ |
| 3. $S ::= R$ | 6. $R ::= L$ |

| | ACTION | | | | GOTO | | |
|------------|--------|-------|-------|-------|------|------|------|
| | id | * | = | eof | S | L | R |
| S_0 | s5/12 | s4/11 | — | — | 1 | 2 | 3 |
| S_1 | — | — | — | acc | — | — | — |
| S_2 | — | — | s6 | r6 | — | — | — |
| S_3 | — | — | — | r3 | — | — | — |
| $S_{4/11}$ | s5/12 | s4/11 | — | — | — | 8/10 | 7/13 |
| $S_{5/12}$ | — | — | r5/12 | r5/12 | — | — | — |
| S_6 | s5/12 | s4/11 | — | — | — | 8/10 | 9 |
| $S_{7/13}$ | — | — | r4/11 | r4/11 | — | — | — |
| $S_{8/10}$ | — | — | r6 | r6 | — | — | — |
| S_9 | — | — | — | r2 | — | — | — |

LALR(1) properties

LALR(1) parsers have same number of states as SLR(1) parsers (core LR(0) items are the same)

In case of *error*, LALR(1) parser may perform more reductions than corresponding LR(1) parser, but will catch error before more input is processed.

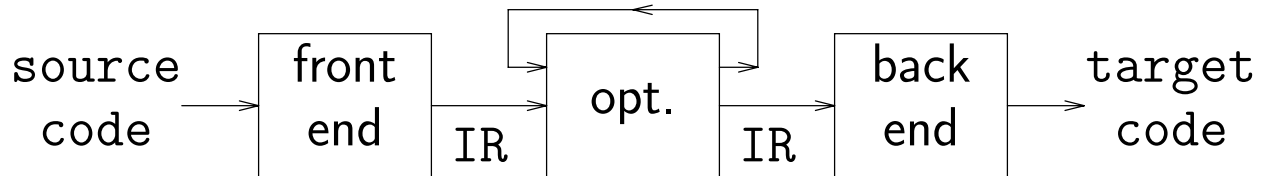
Example grammar with input "id = id =":

$LR(1): S_0 \xrightarrow{s} S_5 \xrightarrow{r} S_2 \xrightarrow{s} S_6 \xrightarrow{s} \mathbf{S}_{12} \Rightarrow \text{error}$

$LALR(1): S_0 \xrightarrow{s} S_{5/12} \xrightarrow{r}$

$S_2 \xrightarrow{s} S_6 \xrightarrow{s} \mathbf{S}_{5/12} \xrightarrow{r} S_{8/10} \xrightarrow{r} S_9 \Rightarrow \text{error}$

FIRST PROJECT: A simple compiler



1. Source code \rightarrow intermediate representation

- (a) storage layout
- (b) code for simple expressions
- (c) code for control structures
- (d) code for procedure calls
- (e) code for complex expressions
- (f) better code for expressions

2. Intermediate representation \rightarrow target code

- (a) instruction selection
- (b) instruction scheduling
- (c) register allocation

We will cover the front-end and some optimizations

First Project

Project has two parts, basic code generation + CSE optimization!

ILOC – intermediate language (quadrupels)

- load-store architecture
- register-transfer language
- three-address code
- explicit loads and stores
- different addressing modes; examples:

| opcode | sources | targets | meaning |
|--------|---------|------------------|-------------------------------|
| load | r1 | \Rightarrow r2 | MEM(r1) \rightarrow r2 |
| loadAI | r1,c1 | \Rightarrow r2 | MEM(r1 + c1) \rightarrow r2 |
| loadAO | r1,r2 | \Rightarrow r3 | MEM(r1 + r2) \rightarrow r3 |

ILOC instruction set to be used for the first project is posted on the project web page.

- **Code shape:** register-register model \Rightarrow freshly created values have their own virtual register; no virtual register is ever redefined!

Storage layout

Local, non-static storage

- stash them in the frame
- keep frames on the stack
- assign offsets from the frame pointer
- pad for word alignment

Global or static storage

- offset from procedure's static data area (static)
- offset from known global label (global)

Varying sized storage

- *local, non-static* \Rightarrow top of stack frame
- *other cases* \Rightarrow allocate on the heap

Storage layout

Key Issue

- what variables can be *safely* allocated to registers?
- what variables *should* be allocated to registers?

Encoding the decision

- assign some variables to virtual registers
- treat those that **cannot** be in a register carefully

Array references

What about $A[i, j]$?

First, we must agree to a storage scheme

row-major order

lay out as sequence of consecutive rows

rightmost subscript varies fastest

$A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]$

column-major order

lay out as sequence of consecutive columns

leftmost subscript varies fastest

$A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]$

Array references

```
integer A[1:10];      // general: A[low:high]
                        // 1 ≤ low ≤ high
x = A[i];            // 1-based indexing
```

How do we compute the address of an array element?

$A[i]$

$$\text{base} + (i - 1) \times w$$

where

- base is relative address (offset) of $A[0]$
- w is $\text{sizeof}(\text{element})$

$$\text{in general: } \text{base} + (i - \text{low}) \times w$$

row-major order, two dimensions

$$\text{base} + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times w$$

column-major order, two dimensions

$$\text{base} + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{low}_1) \times w$$

This looks *expensive!* There are ways to generate more efficient code (hint: algebraic manipulations). See also ALSU pp. 381-384.

Control structures

Assignment statement

$$lhs \leftarrow rhs$$

Strategy

- evaluate rhs to a value (*an rvalue*)
- evaluate lhs to an address (*an lvalue*)
 - i*) *lvalue* is register \Rightarrow **move** it
 - ii*) *lvalue* is address \Rightarrow **store** it

Registers versus memory

- non-aliased scalars \Rightarrow can go in a register
- aggregate or potentially aliased \Rightarrow in memory

Control structures

Basic blocks

- a *basic block* is a sequence of straight line code
- if one instruction executes, they all execute
- a maximal sequence of instructions without branches
- a label starts a new basic block

Early work in code optimization focused on basic blocks.

- common subexpression elimination
- constant folding
- “optimal” code generation
- list scheduling

Control structures

Control structure examples

- `if-then-else`
- `while` loop
- `case` statement

Overview

- control flow links up the basic blocks
- ideas are simple
- implementation requires some bookkeeping
- some care is required for good code
- *design-time* vs. *compile-time* vs. *run-time*

Early optimizing compilers generated good code for basic blocks and linked them together carefully.

Control structures

if-then-else

1. evaluate the expression to **true** or **false**
2. if **true**, fall through to **then** part
branch around **else** part
3. if **false**, branch to **else** part
fall through to next statement

Example

| | |
|-------------------------------|--------------------------------|
| <code>r1 ← expr</code> | <i>evaluate the expression</i> |
| <code>if not(r1) br L1</code> | <i>compare and branch</i> |
| <code>...</code> | <i>stmts for then part</i> |
| <code>br L2</code> | <i>branch to exit</i> |
| <code>L1: ...</code> | <i>stmts for else part</i> |
| <code>L2: ...</code> | <i>following stmt</i> |

Control structures

while loop or do loop

1. evaluate the control expression
2. if **false**, branch beyond end of loop
if **true**, fall through into loop body
3. at end, re-evaluate the control expression
4. if **true**, branch to top of loop body
if **false**, fall through

Example

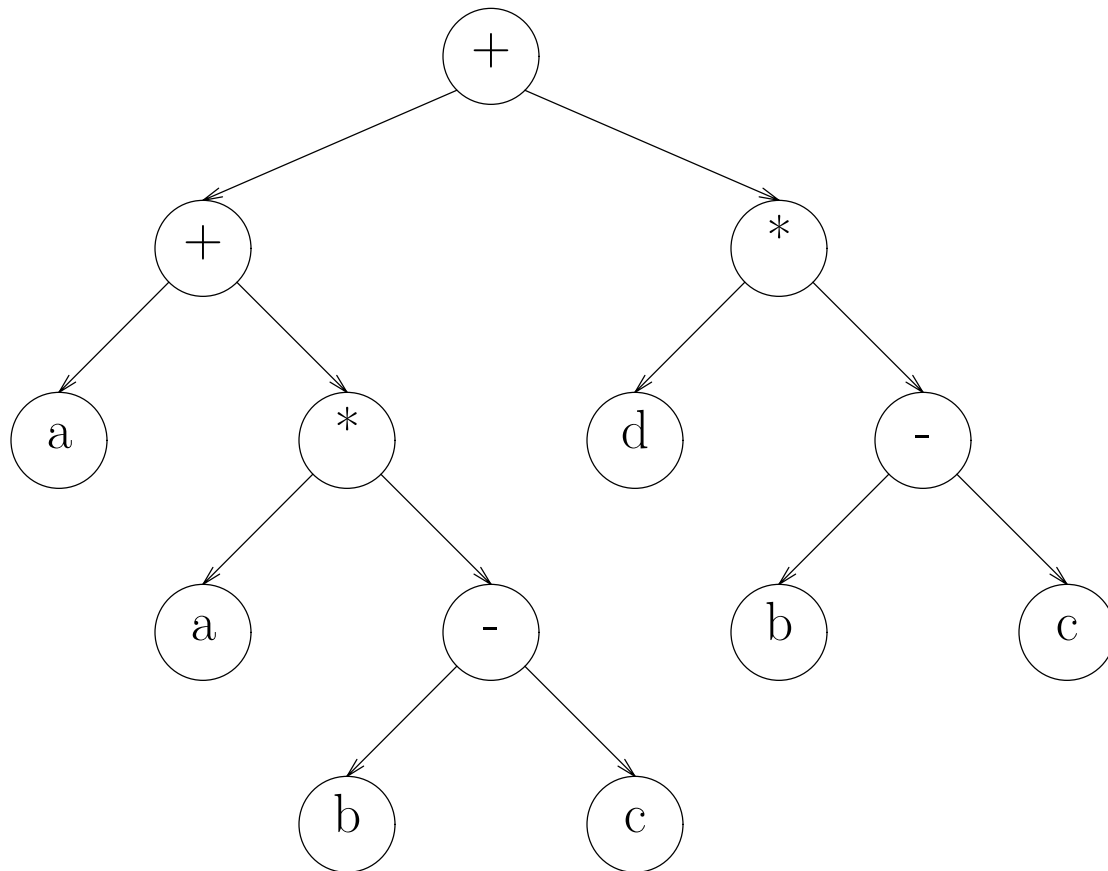
| | | |
|-------------------------------|--|--------------------------------|
| <code>r1 ← expr</code> | | <i>evaluate the expression</i> |
| <code>if not(r1) br L2</code> | | <i>compare and branch</i> |
| L1: <code>...</code> | | <i>loop body</i> |
| <code>r1 ← expr</code> | | |
| <code>if r1, br L1</code> | | |
| L2: <code>...</code> | | <i>following stmt</i> |

Test at end ⇒ simple loop is one block

Part 2: Common subexpressions

Consider the tree for the expression

$$a + a * (b - c) + (b - c) * d$$



Both **a** and **b-c** are common subexpressions (*cse*)

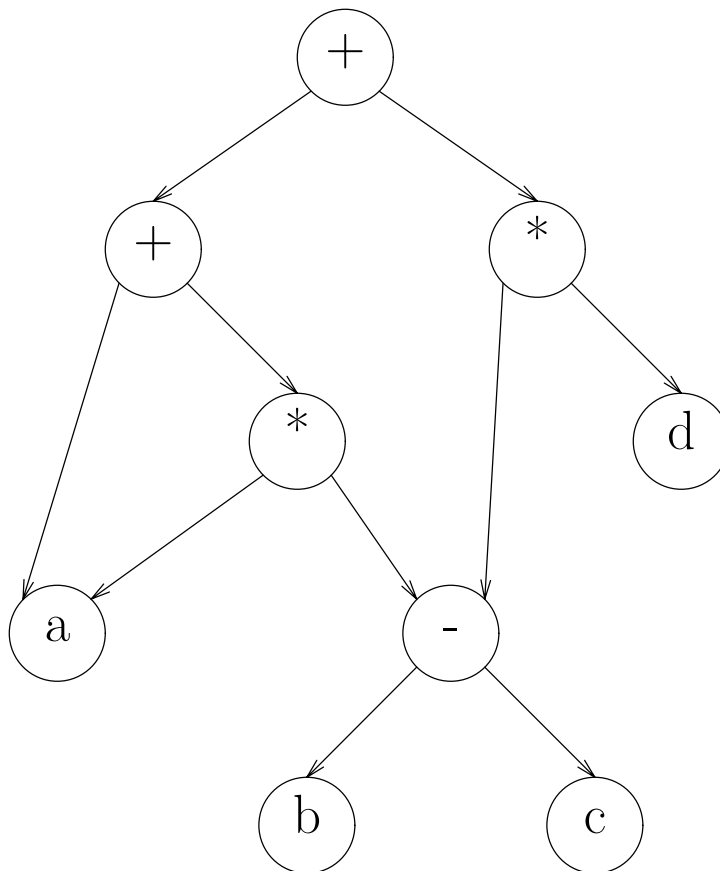
- compute the same value
- should compute the value once

A simple and general form of code improvement

Directed acyclic graphs

The *directed acyclic graph* is a useful representation for such expressions

$$a + a * (b - c) + (b - c) * d$$



The *dag* clearly exposes the *cses*

Aho, Sethi, and Ullman, §5.2, §9.8, ...

Directed acyclic graphs

A *directed acyclic graph* is a tree with sharing

- a tree is a directed acyclic graph where each node has at most one parent
- a *dag* allows multiple parents for each node
- both a tree and a *dag* have a distinguished *root*
- no cycles in the graph!

To find common subexpressions (*within a statement*)

- build the *dag*
- generate code from the *dag*

This should lead to faster evaluation

Directed acyclic graphs

How do we build a *dag* for an expression (single RHS)?

- use construction primitives for building tree
- teach primitives to catch *cse*'s (see (ASU §5.2, ALSU §8.5))
 - *mkleaf()* and *mknnode()*
 - hash on $\langle op, l, r \rangle$
- unique name for each node — its *value number*

Anywhere that we build a tree, we could build a *dag*

- initialize hash table on each expression
- catch only *cse*s within expression

Directed acyclic graphs

What about *assignment* ?

- complicates *cse* detection
- each *value* has a unique node
- add subscripts to variables

While building the *dag*, an assignment $x \leftarrow \dots$

- determine “new” node for *lhs* — a new x_i
- kills all nodes built from x_{i-1}

Example

$$\begin{array}{ll} S_1: & a \leftarrow a + b & \mathbf{a}_1 \leftarrow \mathbf{a}_0 + b_0 \\ S_2: & c \leftarrow a + b & \mathbf{c}_0 \leftarrow \mathbf{a}_1 + b_0 \end{array}$$

How to handle more than a single statement?

Directed acyclic graphs

Use a single dag for an entire basic block

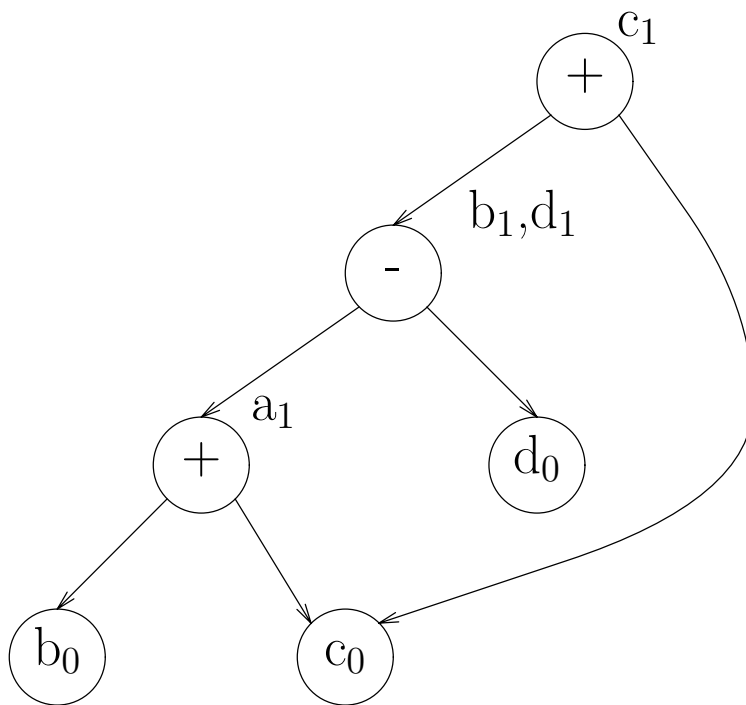
A *dag* for a *basic block* has labeled nodes

1. *leaves are labeled with unique identifier*
 - either variable names or constants
 - leaves represent values on entry, e.g., x_0
2. *interior nodes are labeled with operators*
3. *nodes have optional identifier labels*
 - interior nodes represent computed values
 - identifier label represents assignment

Directed acyclic graphs

Example

| Code | After Renaming |
|----------------------|----------------------------|
| $a \leftarrow b + c$ | $a_1 \leftarrow b_0 + c_0$ |
| $b \leftarrow a - d$ | $b_1 \leftarrow a_1 - d_0$ |
| $c \leftarrow b + c$ | $c_1 \leftarrow b_1 + c_0$ |
| $d \leftarrow a - d$ | $d_1 \leftarrow a_1 - d_0$ |



Let's assume our basic code generation algorithm (generate code for left child subtree, then right child subtree, then root of subtree). How does the code look like?

Directed acyclic graphs

Building a dag

node($\langle id \rangle$) \rightarrow current *dag* for $\langle id \rangle$

1. set node(y) to undefined, for each symbol y
2. for each statement $x \leftarrow y \text{ op } z$, repeat steps 3, 4, and 5
3. if node(y) is undefined,
 create a leaf for y
 set node(y) to the new node
do the same for z
4. if $\langle \text{op}, \text{node}(y), \text{node}(z) \rangle$ doesn't exist,
 create it;
 let n be the node found or newly created node
5. delete x from the list of labels for node(x)
 append x to the list of labels for n
 set node(x) to n

Aho, Sethi, and Ullman, Algorithm 9.2, in §9.8

Directed acyclic graphs

Reality

Do compilers really use this stuff?

The *dag* construction algorithm is fast enough

A compilers that uses quads will (*often*)

- build a *dag* to find *cses*
- convert back to quads for later passes

Are there many *cses*? *Yes!*

- they arise in addressing
- array subscript code
- field access in records
- expressions based on loop indices
- access to parameters

First Project, Part 2

Perform CSE (common subexpression elimination) across entire basic blocks.

How does the DAG look like for the following basic block?

$a = c + 1$

$c = b + 2$

What's the problem here? \Rightarrow **anti-dependence**

Can we avoid this problem if we (1) perform CSE on “ILOC instructions” and (2) assume a particular code shape, namely that every new value is assigned a new virtual register.

Attribute grammars

Formal framework based on grammar and parse tree

Idea: attribute the tree

- can add attributes (*fields*) to each node
- specify equations to define values (*unique*)
- both inherited and synthesized attributes

Attribute grammars are very general. Can be used for

- infix to postfix translation of arithmetic expressions
- type checking (*context-sensitive analysis*)
- construct intermediate representation (*AST*)
- desk calculator (*interpreter*)
- code generation (*compiler*)

Attribute grammars

Aho, Sethi, & Ullman describe *syntax-directed definitions*. These are just *attribute grammars* by another name.

Attribute grammar

- generalization of context-free grammar
- each grammar symbol has an associated set of attributes
- augment grammar with rules defining attribute values. Semantic rules may have side effects!
- high-level specification, independent of evaluation scheme (*Note: translation scheme* has eval. order).

Dependencies between attributes

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

Attribute grammars

Note:

- terminals can be associated with values returned by the scanner. These input values are associated with a synthesized attribute.
- distinguished nonterminal (*starting symbol*) cannot have inherited attributes.
- synthesized attributes of a grammar symbol can depend on inherited attributes of the same symbol.
- semantic rules are defined for each production separately
⇒ attribute dependence edges go only 1 level in the parse tree.
- semantic rules associated with a rule $A ::= \alpha$ have to specify the values for all
 - synthesized attributes for A (root)
 - inherited attributes for grammar symbols in α (children)

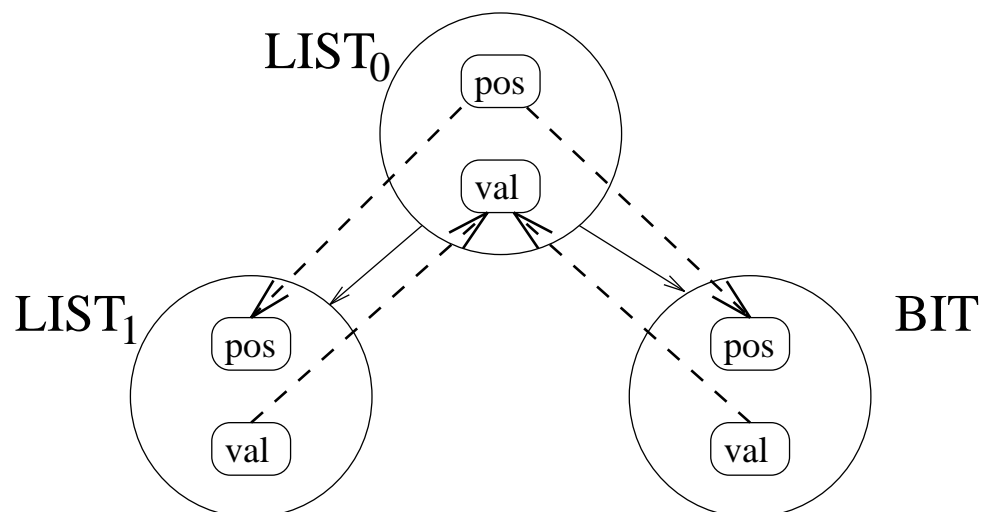
Example attribute grammar

A grammar to evaluate signed binary numbers

| Production | Semantic Rules |
|---|---|
| 1 NUM ::= SIGN LIST | LIST.pos \leftarrow 0 NUM.val \leftarrow if SIGN.neg then -LIST.val else LIST.val |
| 2 SIGN ::= + | SIGN.neg \leftarrow false |
| 3 SIGN ::= - | SIGN.neg \leftarrow true |
| 4 LIST ::= BIT | BIT.pos \leftarrow LIST.pos LIST.val \leftarrow BIT.val |
| 5 LIST ₀ ::= LIST ₁ BIT | LIST ₁ .pos \leftarrow LIST ₀ .pos + 1 BIT.pos \leftarrow LIST ₀ .pos LIST ₀ .val \leftarrow LIST ₁ .val + BIT.val |
| 6 BIT ::= 0 | BIT.val \leftarrow 0 |
| 7 BIT ::= 1 | BIT.val \leftarrow 2 ^{BIT.pos} |

Example attribute grammar

| Production | Semantic Rules |
|---------------------------|---|
| 5 $LIST_0 ::= LIST_1 BIT$ | $LIST_1.pos \leftarrow LIST_0.pos + 1$ $BIT.pos \leftarrow LIST_0.pos$ $LIST_0.val \leftarrow LIST_1.val + BIT.val$ |



source $-- \Rightarrow$ sink dependency

Note:

- semantic rules define partial *dependency graph*
- structure can be used to derive characteristics of generated total dependency graphs.

Attribute grammars

The attribute dependency graph

- nodes represent attributes
- edges represent the 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*

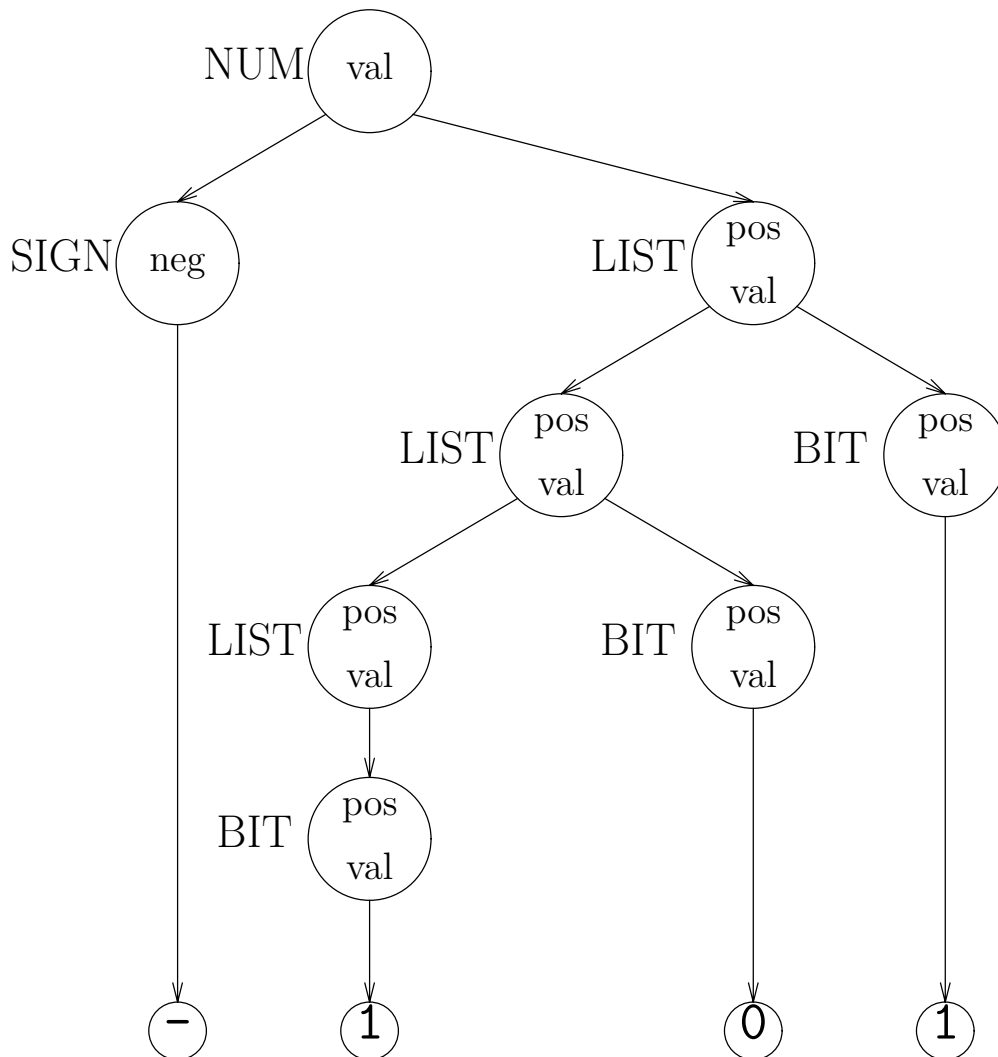
Evaluation order

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

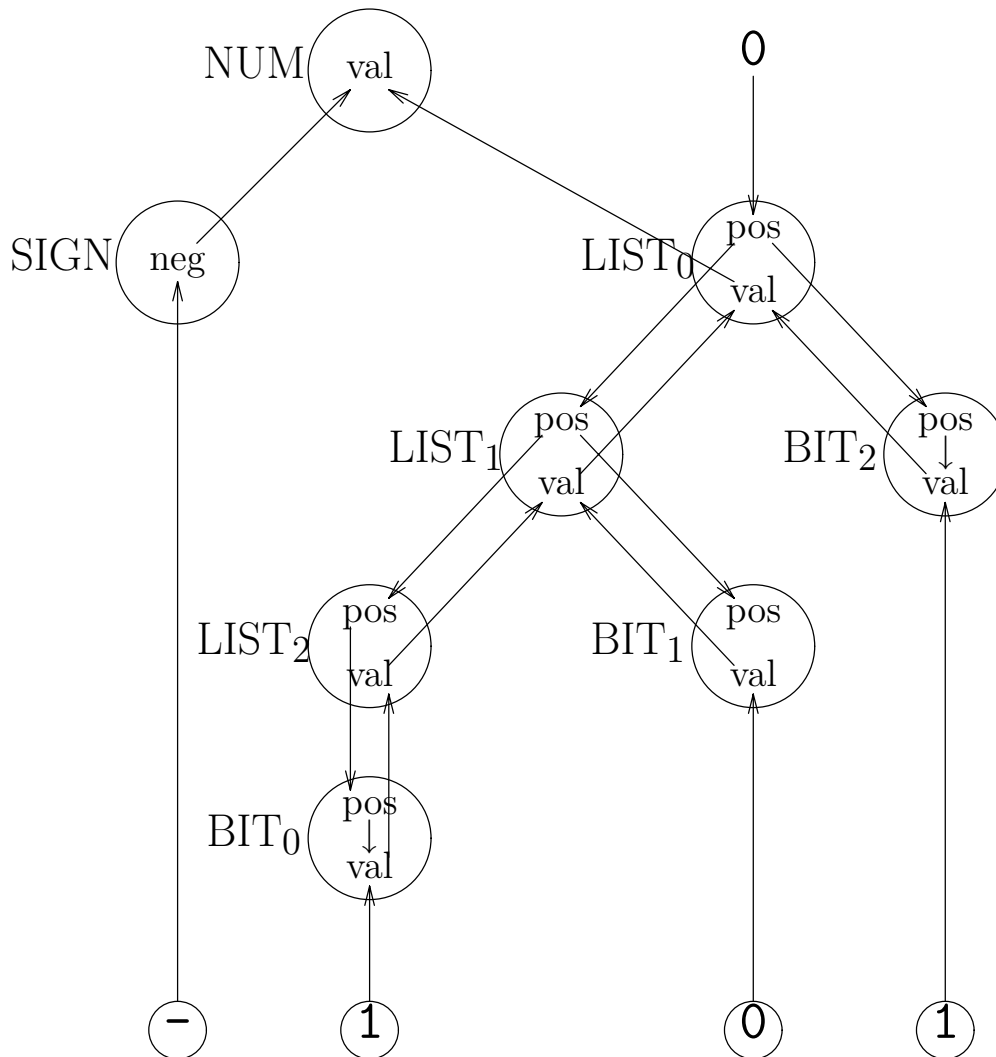
This order depends on both the grammar and the input string.

Example attribute grammar

Example Parse tree for -101



Example grammar dependence graph



- `val` and `neg` are *synthesized* attributes
- `pos` is an *inherited* attribute

LIST₀.pos is an inherited attribute with an empty dependency set.

Attribute grammars

A topological order for the example

1. SIGN.neg
2. LIST₀.pos
3. LIST₁.pos
4. LIST₂.pos
5. BIT₀.pos
6. BIT₁.pos
7. BIT₂.pos
8. BIT₀.val
9. LIST₂.val
10. BIT₁.val
11. LIST₁.val
12. BIT₂.val
13. LIST₀.val
14. NUM.val

Evaluate in this order

Yields NUM.val: -5

Classification of evaluation methods

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. analyze rules at compiler-generation time
2. determine a static ordering at that time
3. evaluate nodes in that order at compile time

ad-hoc methods (passes, dataflow)

1. ignore the parse tree and grammar
2. choose a convenient order and use it
(forward-backward passes, alternating passes)

Problems

- circularity
- best evaluation strategy is grammar dependent

Attribute grammars

Advantages

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

Disadvantages

- evaluation strategy determines efficiency
- increased space requirements
- results distributed over tree
- circularity testing

We will use a **syntax-directed translation** approach which is similar in spirit to an attribute grammar (“keep is local, stupid”), but allows access to global variables.

Watch out - There is no check whether access to local data is appropriate or not (e.g.: only access data when it is actually allocated in the stack).

Lambda calculus

- formalism for studying ways in which functions can be formed, combined, and used for computation
- **computation** is defined as rewriting rules (operational semantics)
- the syntactic notion of computation was developed first; a mathematical semantics followed much later

Examples:

| | | |
|-------------------------------|-----------------|---|
| $f(x) = x+2$ | $\lambda x.x+2$ | different notation |
| $(\lambda x.x+2) 1$ | $1+2 = 3$ | function application and substitution |
| $(\lambda x.x) (\lambda y.y)$ | | arguments and returned “values” can be functions |
| $\lambda x.xx$ | | untyped lambda calculus $f(x) = x(x)$ |

Lambda calculus and functional programming

Lambda calculus is the theoretical foundation of pure functional programming (no side effects, *referential transparency*)

Functional programming: functions are *first class citizens*

- can be a return value
- can be passed as arguments
- can be put into a data structure
- value of an expression can be a function

$((\lambda x.x) (\lambda x.1)) (\lambda y.y)$

Currying functions

$f : D^n \rightarrow D$ can be transformed to

$$f : \underbrace{D \rightarrow (D \rightarrow \dots (D \rightarrow D)) \dots}_{n\text{-times}}$$

Examples:

$$\begin{array}{ll} f(x, y) = x + y & f : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N} \\ f'(x) = g_x(y) & f' : \mathcal{N} \rightarrow (\mathcal{N} \rightarrow \mathcal{N}) \\ & g_x : \mathcal{N} \rightarrow \mathcal{N} \\ & g_x(y) = x + y \end{array} \qquad \begin{array}{l} g_2 : \mathcal{N} \rightarrow \mathcal{N} \\ g_2(y) = 2 + y \end{array}$$

$g_x(y)$ “freezes” the first argument value x .

This is sometimes referred to as **partial evaluation**.

Here is an example:

$$\begin{aligned} (\lambda x. \lambda y. x + y) 2 3 &= \\ (\lambda y. 2 + y) 3 &= \\ 2 + 3 &= 5 \end{aligned}$$

To simplify discussion, all n -ary functions are curried, i.e., are represented by a sequence of one-place functions.

Pure Functional Languages and Scheme

For next time, please read: Scott: Chapter 10