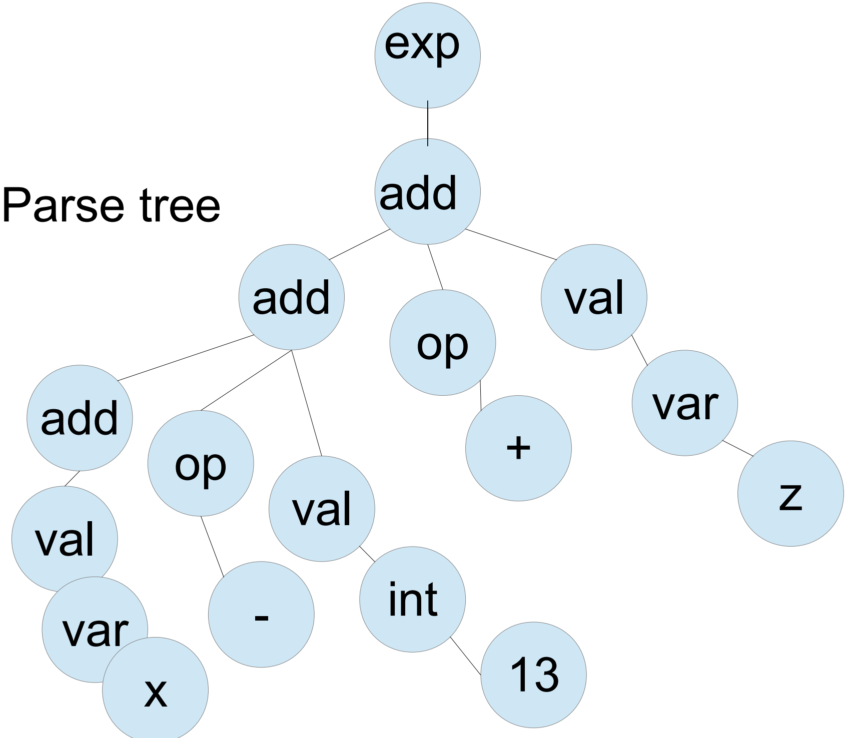
# IR trees/graphs

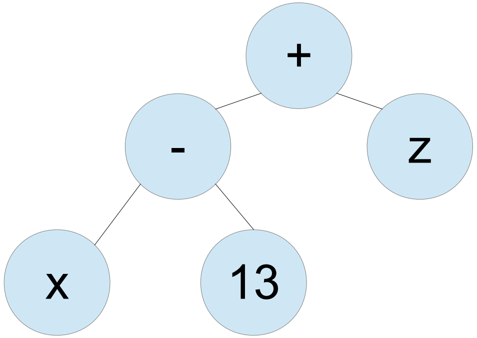
- Various possible tree/graph intermediate representations:
- Parse tree: directly based on grammar of the source language
- Syntax tree: abstract from parse tree (less language dependent)
- Dependency graph: show heirarchy of declared/defined items, and which ones depend on which others
- Control flow graphs: divide code into uninterrupted blocks of code, with directed edges indicating possible flow between them
- Call graphs: nodes for procedures, directed edges indicate calls

# Parse trees vs syntax trees

- Attempt to abstract data and operations away from language specific grammar rules

Parse tree

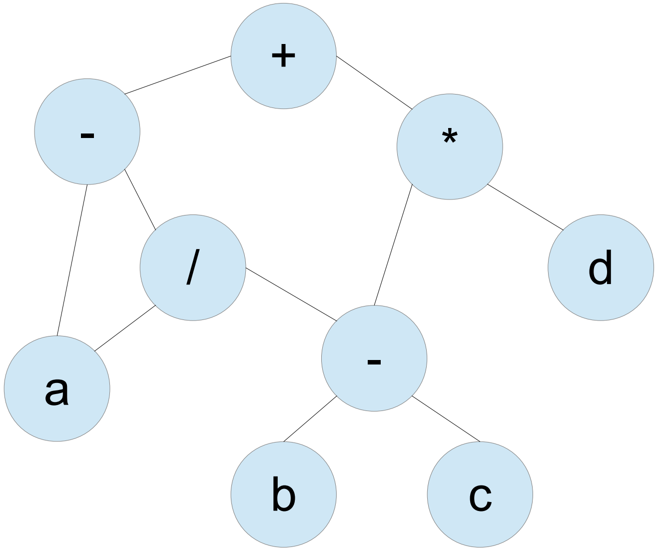



Abstract idea

# Directed acyclic syntax graphs

- Reduce size by identifying/reusing common subtrees
- (a-(a/(b-c))) + ((b-c)*d)



# Syntax DAGs

- Need an effective way to recognize when common subtrees exist, ideally asap during construction
- Introduces many possible language-independent optimizations, e.g.
  - in expressions: store result of subtree in a temp variable rather than recomputing
  - if subtree represents a block of statements then replace with callable function)

# Tree construction from grammar

- Suppose we build our tree with two kinds of nodes:
  - leaf: holds the tokens in our source grammar
  - node: internal node, corresponding to nonterminals
- For each grammar rule, we define a rule on how to build the appropriate tree node/leaf
- For top-down derivation, we start with a node for our top level nonterminal, and on each rule application we apply the appropriate construction

# Example: construction rules

| Rule | Construction |
|---|---|
| expr-->addx | expr.node = addx.node |
| addx--> addx aop multx | addx.node = new node(aop.node, addx.node, multx.node) |
| addx --> multx | addx.node = multx.node |
| multx --> multx mop valx | multx.node = new node(mop.node, multx.node, valx.node) |
| multx --> valx | multx.node = valx.node |
| valx --> VAR \| NUM | valx.node = new leaf (VAR, VAR.txt)<br>valx.node = new leaf (NUM, NUM.val) |
| valx --> '(' expr ')' | valx.node = new node( '(', expr.node, ')' ) |
| aop --> '+' \| '-' | aop.node = new leaf('+')<br>aop.node = new leaf('-') |
| mop --> '*' \| '/' | mop.node = new leaf('*')<br>mop.node = new leaf('/') |

# Array-of-records implementation

- Need a way to represent our leaf/node collection, e.g.
  - leaf record type
  - node record type
  - keep an array of records (and counter)
- Each leaf/node thus has a unique index value (array pos)
- Cross references between nodes can use the index (giving small storage, fast lookups)
- Often referred to as value-number method, each node has unique associated index number

# Value-number example

i = i + x * 10
i.e.
i = (i + (x * 10))

| index | Node/leaf | data1 | data2 |
|---|---|---|---|
| 0 | (leaf x) | symtable ptr for x | |
| 1 | (leaf i) | symtable ptr for i | |
| 2 | (leaf 10) | literal 10 | |
| 3 | (node *) | 0 (index of node x) | 2 (index of node 10) |
| 4 | (node +) | 1 (index of node i) | 3 (index of node *) |
| 5 | (node =) | 1 (index of node i) | 4 (index of node +) |

# Searching problem:

- As we're building the array, we need to search current array content to find operand indices, e.g. to fill in fields for x * 10 we need to find indices for nodex x and 10
- Currently that means a linear search: O(n)
- Could store the nodes as a binary search tree instead of an array, so O(log(n))
- Could store the nodes in hash table: collection of buckets, with hash function mapping the operands (e.g. *, X, 10) to a bucket, then just linear search the bucket if not empty

# Using duplicate subtrees (DAG)

- When building an entry, and have searched for the correct operand indices, look at fields for new entry, check if there's already a matching entry
- e.g. Suppose we have a new entry using x*10 again, we look for a (node *) with data fields 0 and 2, and find index 3 already provides it

| index | Node/leaf | data1 | data2 |
|---|---|---|---|
| 0 | (leaf x) | symtable ptr for x | |
| 1 | (leaf i) | symtable ptr for i | |
| 2 | (leaf 10) | literal 10 | |
| 3 | (node *) | 0 (index of node x) | 2 (index of node 10) |
| 4 | (node +) | 1 (index of node i) | 3 (index of node *) |
| 5 | (node =) | 1 (index of node i) | 4 (index of node +) |