ELEC 875
Design Recovery
and
Automated Evolution

Week 2 Class 2
Context Free Grammars
and Parsing
Use in Models

Next Class Reading

- T. Lethbridge, E. Plödereder, S. Techelaar, C. Riva, P. Linos, S. Marchenko, “The Dagstuhl Middle Model”
  - ◊ DMM Schema
  ->http://www.site.uottawa.ca/~tcl/dmm/DMMDescriptionV0006.pdf

- H. Fahmy, R.C. Holt and J.R. Cordy, "Wins and Losses of Algebraic Transformations of Software Architectures", Proc. ASE'2001, IEEE 16th International Conference on Automated Software Engineering, San Diego, November 2001, pp. 51-62.

Overview

- Scanning vs. Parsing
- Context Free Grammars
- TXL
- Languages and Language Features

Scanning vs Parsing

- Compilers and most other language analysis operates at two levels.
- Scanning - token level processing
- Parsing - tree level processing

Scanning

- Lexical Analysis
- Tokens can be described as Regular Expressions
- Separate the input into tokens
- In most languages, scanning is separate from parsing - scanner is called as a co-routine.
- Issues
  - ◊ Some languages change scan rules on instruction from the parser.
    - Perl
    - Embedded languages (SQL inside of COBOL)
  - ◊ spaces, comments, file boundaries can be important

Scanning - embedded languages

```
if ($abc =~ /foo/)

if ($abc =~ /foo|bar*/)
```

Scanning - embedded languages

```
01 NAME        PIC X(20).
01 HRS         PIC 999.
01 DEPARTMENT  PIC X(20).
01 EMPNO       PIC 999999.

MOVE 810153 TO EMPNO.
EXEC SQL
    SELECT NAME, HOURS, DEPT
        INTO :NAME, :HRS, :DEPARTMENT
        FROM EMPLOYEE
        WHERE EMPNO = :EMPNO
END-EXEC
….
```

Scanning - embedded languages

```
PreparedStatement stmt = conn.prepareStatement(
    "SELECT NAME, HOURS, DEPT"
  + " SELECT NAME, HOURS, DEPT"
  + " WHERE EMPNO = ?");
stmt.setBigDecimal(810153, salary);
rs = stmt.executeQuery();
if (!rs.next()) {
}
```

```
empno = 810153;
#sql {
    SELECT NAME, HOURS, DEPT
        INTO :name, :hrs, :department
        SELECT NAME, HOURS, DEPT
        WHERE EMPNO = :empno
}
```

Scanning Example

int main(int argc,char *argv)

Tokens:

identifier	“int”	star	
space	“ “	identifier	“argv”
identifier	“main”	close bracket	
open bracket		newline	
identifier	“int”		
space	“ “		
identifier	“argc”		
comma			
identifier	“char”		
space	“ “		

Context Free Grammars

- Context free grammar is a 4 tuple:

  (V_T, V_N, S, P)

  where:

  V_T is a finite set of terminal symbols (tokens)

  V_N is a finite set of non-terminal symbols

  $S \rightarrow V_N$ is the start symbol

  P is a set of rules or productions of the form

  $A \rightarrow \alpha$

  where

  $A \in V_N$

  $\alpha \in (V_N \cup V_T)^*$

Example

- Simple Precedence Expressions

V_T = { id, number, +, -, *, /, (,) }

V_N = { E, T, F }

S = E

P = $E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

$F \rightarrow number$

Derivation of Sentences

- A Sentence of the grammar is a sequence of terminal symbols that is derivable from the start symbol and productions
- Start at goal symbol and replace elements of V_N using one of the productions.
- Each step is a derivation
- Done when all of the symbols are terminal symbols

Example Derivation

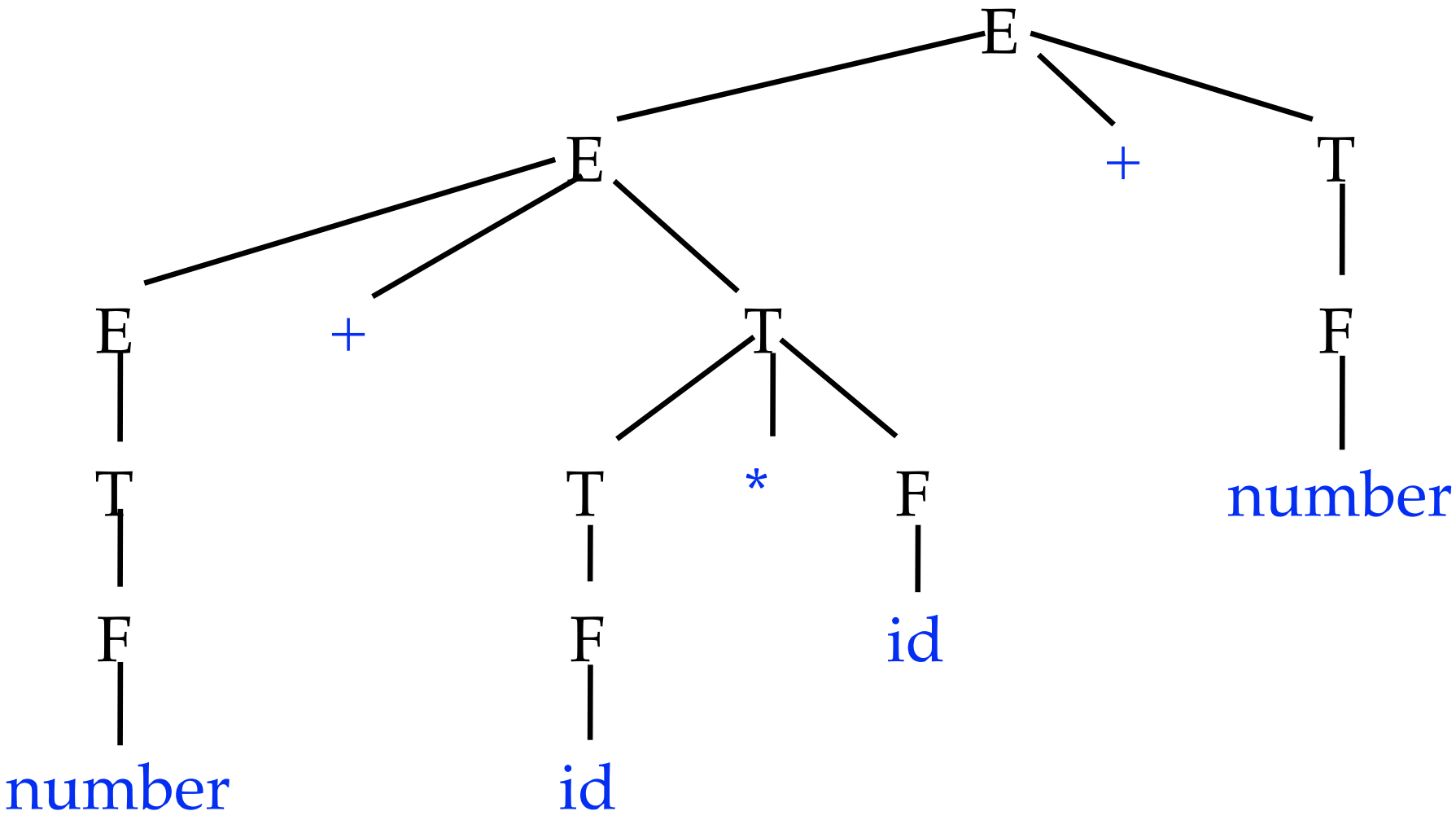
E	E → E + T
E + T	E → E - T
E - T + T	E → T
T - T + T	T → F
F - T + T	F → number
number - T + T	T → T * F
number - T * F + T	T → F
number - F * F + T	F → id
number - id * F + T	F → id
number - id * id + T	T → F
number - id * id + F	F → number
number - id * id + number	

Notes

- some tokens recognized as token *classes*
  - ◊ id, number
  - ◊ value of token is an attribute
- Leftmost Derivation
  - ◊ leftmost symbol of each *sentential form* is replaced
  - ◊ what is a rightmost derivation?
- Grammar is Left Recursive
  - ◊ problem for top down parsers
    - TXL has heuristic to fix Left Recursive Grammars
  - ◊ Right Recursive?

Parse Trees

- graph representation of derivations



Parsing

- Construct the derivation for a given input string
- If there is more than one parse tree for a given input, the parse is ambiguous
  - ◊ ambiguity can be useful
- For modern languages, parse trees reflect the structure of the program
  - ◊ Contents of a function are subtrees within the parse tree of the function
- Compiler grammars may not be appropriate
  - ◊ optimized for semantic analysis and code generation
  - ◊ optimized for speed for the parser implementation

Example

Program → { VarDecl | Function | TypeDecl }

VarDecl → TypeName VarList ';'

Function → [TypeName] identifier FunctionHeader
Block

VarList → identifer { ',' VarList}

TypeName → void | int | char | float | identifier

Example (cont'd)

FunctionHeader → '(' [ParmDecl { ',' ParmDecl }] ')'

ParmDecl → TypeName identifier

Block → '{' { VarDecl | TypeDecl } { Stmt } '}'

Stmt → IfStmt | AssignStmt | ProcCall | ... | Block

IfStmt → if '(' Expr ')' Stmt ['else' Stmt]

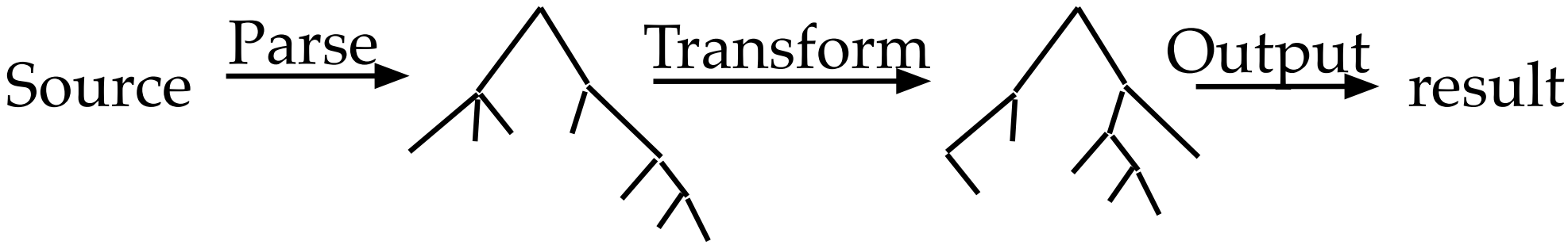
TXL

- functional language
- grammar programming
- strongly typed language

- A TXL program consists of two parts
  - ◊ grammar
  - ◊ rules

TXL

- 3 stages
  - ◊ parse input (result is tree)
  - ◊ run rules (change tree)
  - ◊ generate output (unparse)



TXL Grammar

- goal symbol is the symbol ‘program’

```
define program
    [repeat element]
end define

define element
    [varDecl] | [typeDecl]  | [function]
end define

define function
    [opt typeName] [id] [header] [body]
end define
```

TXL Grammar

- grammar can be changed

```
include “Java.grammar”

redefine statement
    ...
  | [sqlj _statment]
end redefine
```

TXL Rules

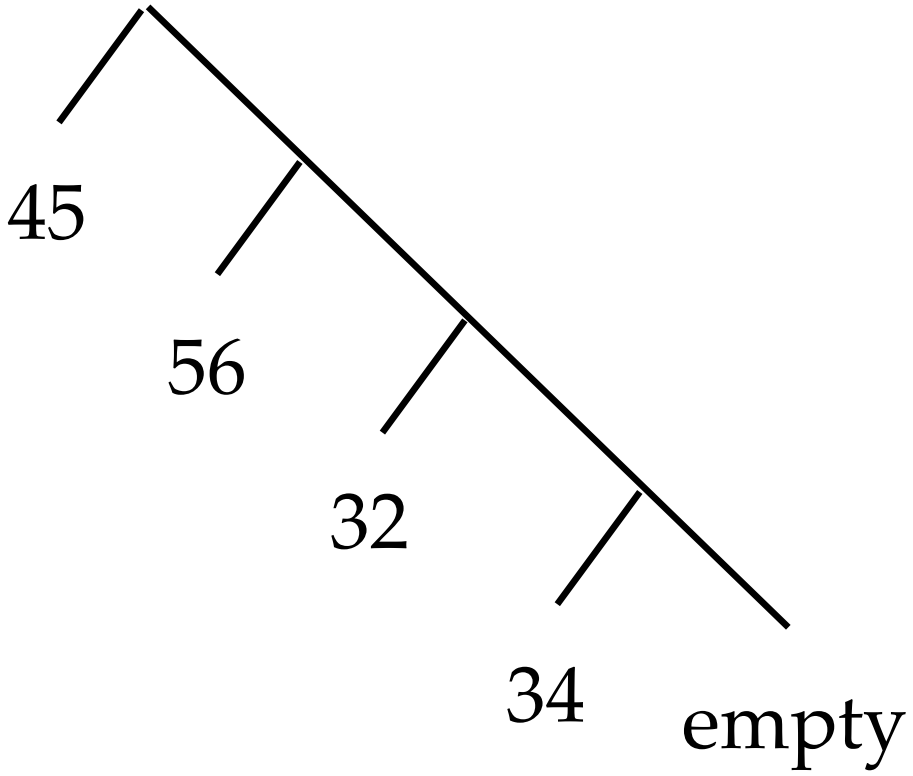
- rule has a pattern and a replacement
  - ◊ search for pattern, replace by replacement
  - ◊ may call sub-rules

```
define program
    [repeat number]
end define

rule main
    replace [repeat number]
        N1 [number] N2 [number]
        Rest [repeat number]
    by
        N1 [+ N2] Rest
end rule
```

TXL Rules

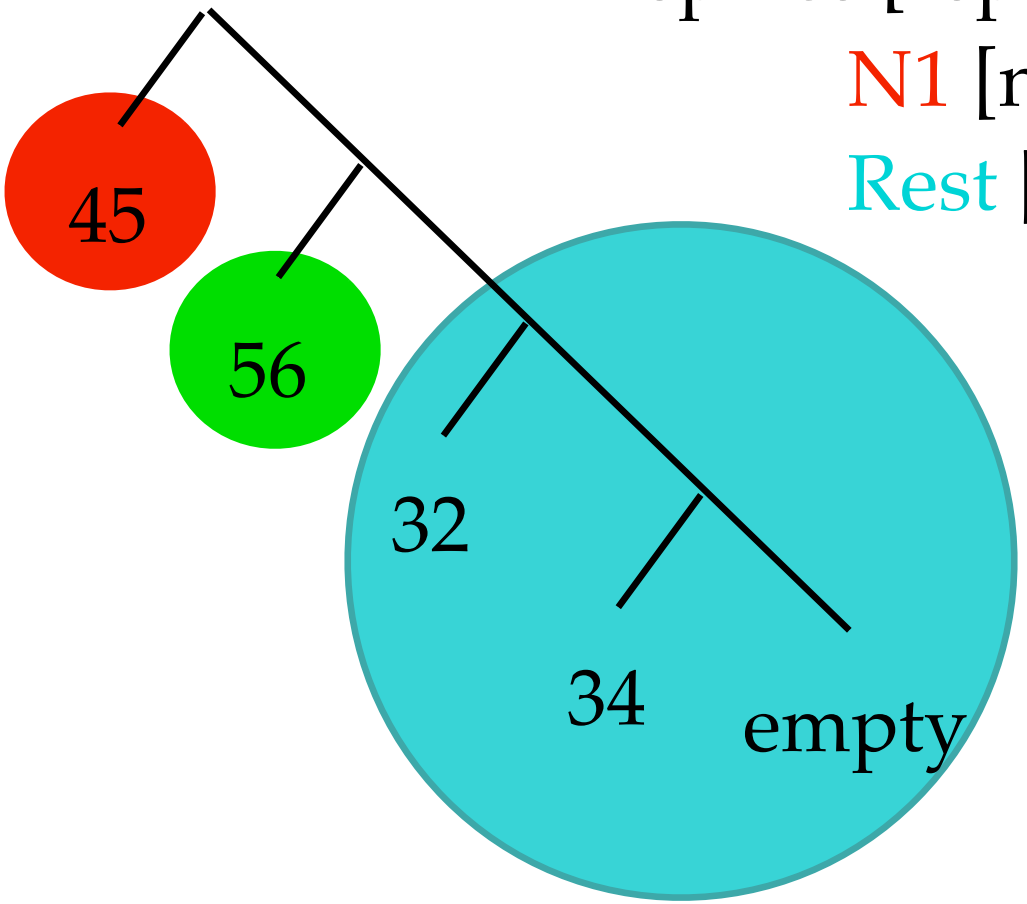
Input: 45 56 32 34



TXL Rules

Input: 45 56 32 34

replace [repeat number]
N1 [number] N2 [number]
Rest [repeat number]

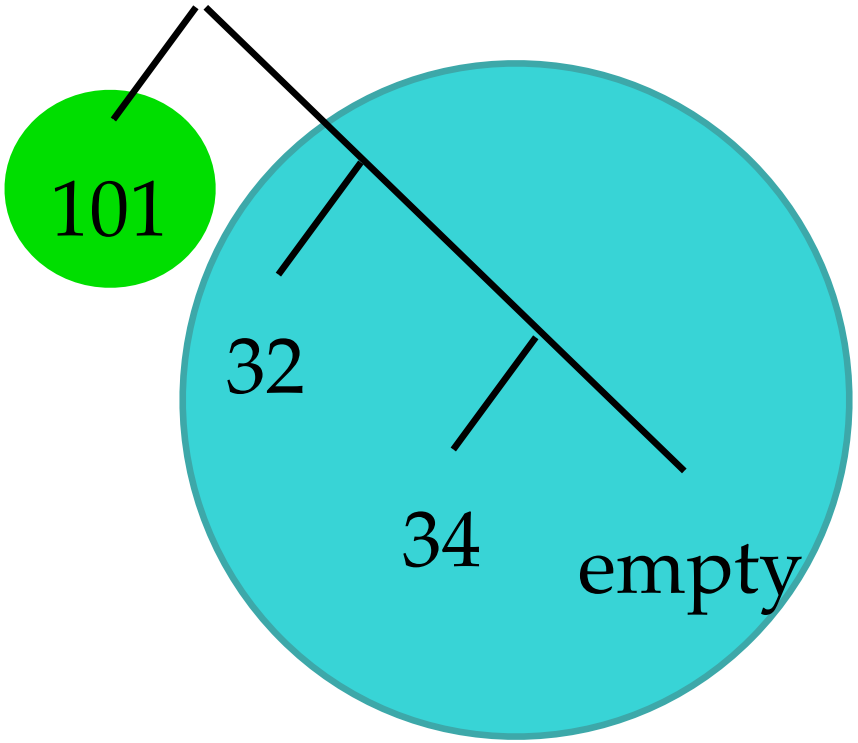


TXL Rules

Input: 45 56 32 34

by

N1 [+ N2] Rest



TXL Rules

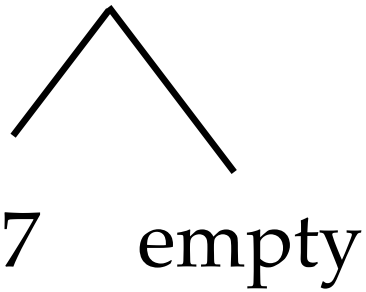
- patterns must be parsable by the grammar
  - ◊ construct partial tree

```
define program
    [repeat number]
end define

rule main
    replace [repeat number]
        N1 [number] N2 [number]
        Rest [repeat number]
    by
        N1 [+ N2] Rest
end rule
```

TXL Rules

- pattern fails because there is only one number, pattern requires two numbers
- pattern fails means program stops, and th tree is output



- result: 167

TXL Functions

- like rules
  - ◊ only apply once
  - ◊ apply only at top of tree (except searching functions)

```
function fixFortranSubscript
    replace [varRef]
        ArrayName [id] ( N [number] + V [id] )
    by
        ArrayName ( V + N )
end rule
```

TXL Unification

- variables can place constraints on match

```
function optimizeAssign
    replace [assignment]
        V [id] = V + E [expression]
    by
        V += E
end rule
```

Deconstruct

- refine patterns
  - ◊ allow to pull apart subtrees matched in main pattern

```
function fixFortranSubscript
    replace [varRef]
        ArrayName [id] (  Sub [subscript] )
    deconstruct Sub
        N [number] + V [id]
    by
        ArrayName ( V + N )
end rule
```

Where

- condition on values

```
function optimizeAssign2
    replace [assignment]
        Var [id] += N [number]
    where
        N [= 1]
    by
        V ++
end rule
```

TXL Notes

- grammar is flexible. Can make changes specific to the program you are writing
  - ◊ Let the parser do the work!!
  - ◊ Multiple passes, where each pass has a slightly different grammar

- txl documentation
  - ◊ www.txl.ca
  - ◊ txl challenge

Languages

- Top Languages (numbers are estimates)
  - ◊ COBOL
    - 500 billion to 1.5 trillion lines in 1998 (depends on who you listen to)
    - ~ 60-65% of existing code base
    - 5 billion more lines by next year
  - ◊ PL/I
    - ~ 5% of existing code base
  - ◊ RPG
    - ~ 5% of existing code base
  - ◊ rest is all other languages

Language features

- variable declarations
  - ◊ type, scope, storage layout
  - ◊ int x;
  - ◊ 05 X PIC 99V99.
  - ◊ structured vars (COBOL, PL/I)

- type definitions
  - ◊ simple types (typedef char * foo)
  - ◊ compound types (records, structs, classes)
    - slack bytes
  - ◊ anonymous type definitions
    struct { … } foobar

Language features

- functions
  - ◊ return type
  - ◊ parameters
    - type, reference, value, name, value-result
    - type conversions
  - ◊ calls to functions, arguments

- statements
  - ◊ complete model?
  - ◊ simplified model

    MOVE A TO B, C

    A = B + C

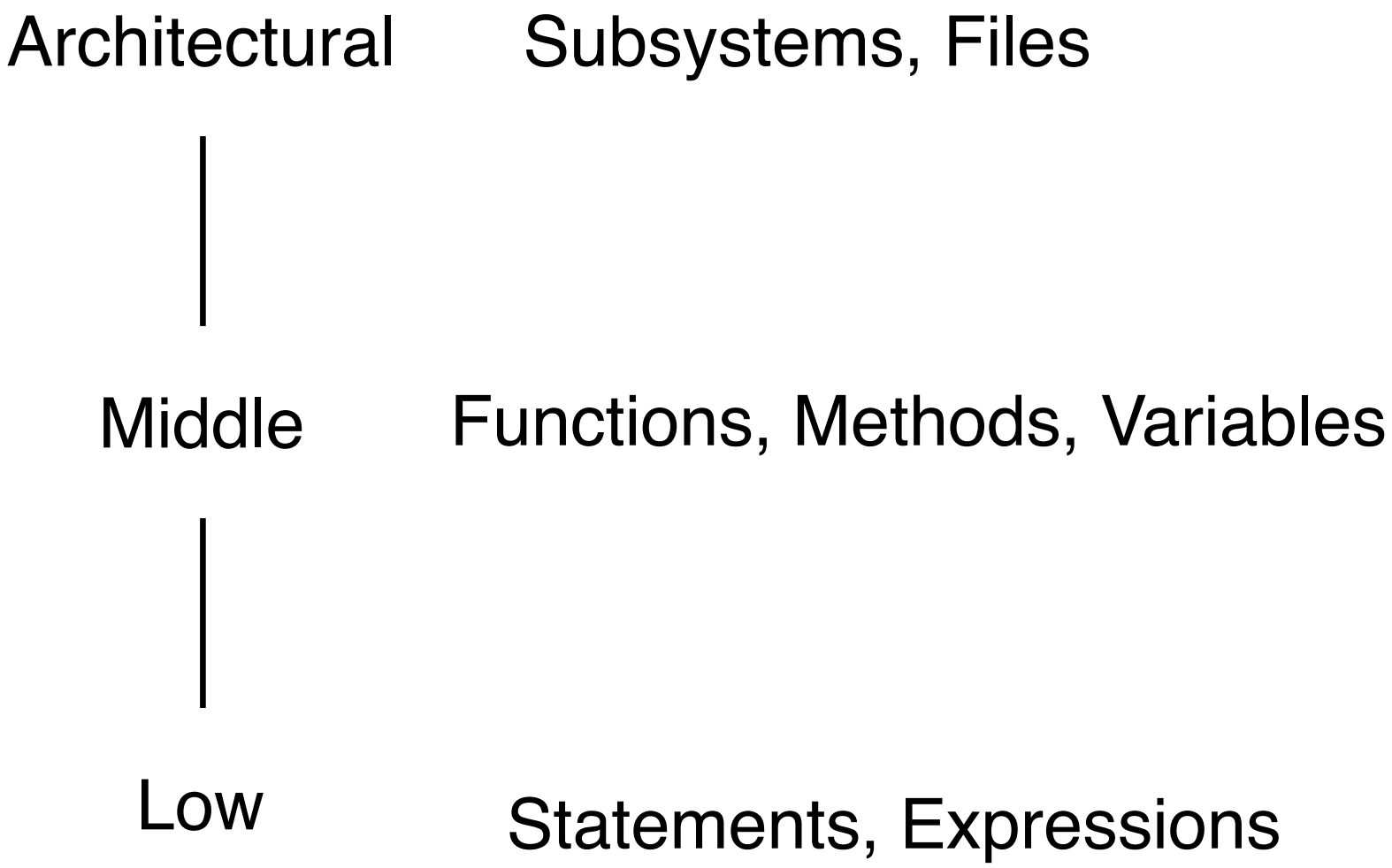
Language features

- expressions
  - ◊ types
  - ◊ type conversions

- variable uses
  - ◊ read/modify
  - ◊ role (subscript?)
  - ◊ values?

- I/O
  - ◊ Languages with I/O (COBOL, PL/I)
  - ◊ indexed files, key values

Model Levels



Towards a Std. Schema for C/C++

- several existing schemas
  - ◊ Datrix/CPPX
  - ◊ Columbus

- Separation of Tools
  - ◊ Everyone has to write an extractor
  - ◊ little research in new extractors (overhead)

- Complete Schemas
  - ◊ full parse tree
  - ◊ tool extracts information
  - ◊ easier to extract information from database (?)

Datrix

- Bell Canada
  - ◊ Datrix Project
  - ◊ C/C++/Java
  - ◊ Templates only partially supported
  - ◊ CPPX implementation

- Source Complete
  - ◊ redundant parens eliminated
  - ◊ CPPX is not source complete, but source equivalent

Columbus

- University of Szeged
  - ◊ Source Complete - but no redundant parens
  - ◊ Recently complete
  - ◊ C/C++

Representation

- Lexical
  - ◊ preprocessing not modelled
  - ◊ line/columns
  - ◊ multiple files (mangle/namespace)

- Syntax
  - ◊ AST - generate code by walking AST
    - – not completely true in both cases
    - – types are refers edges
    - – difficulties with templates

Representation

- Syntax
  - ◊ Datrix is based on semantic model of types
  - ◊ Columbus is based on syntactic model of types
  - ◊ tradeoffs?

- Statements
  - ◊ both models completely model statements now

Representation

- Naming
  - ◊ each entity in a database has to have some unique identifier
  - ◊ Both use arbitrary numbers as identifiers
  - ◊ names of entities are attributes
  - ◊ C++ style mangles to link models

- Currently no closer to a standard model
  - ◊ CPPX (Datrix) was used in Waterloo SWAG project

Datrix

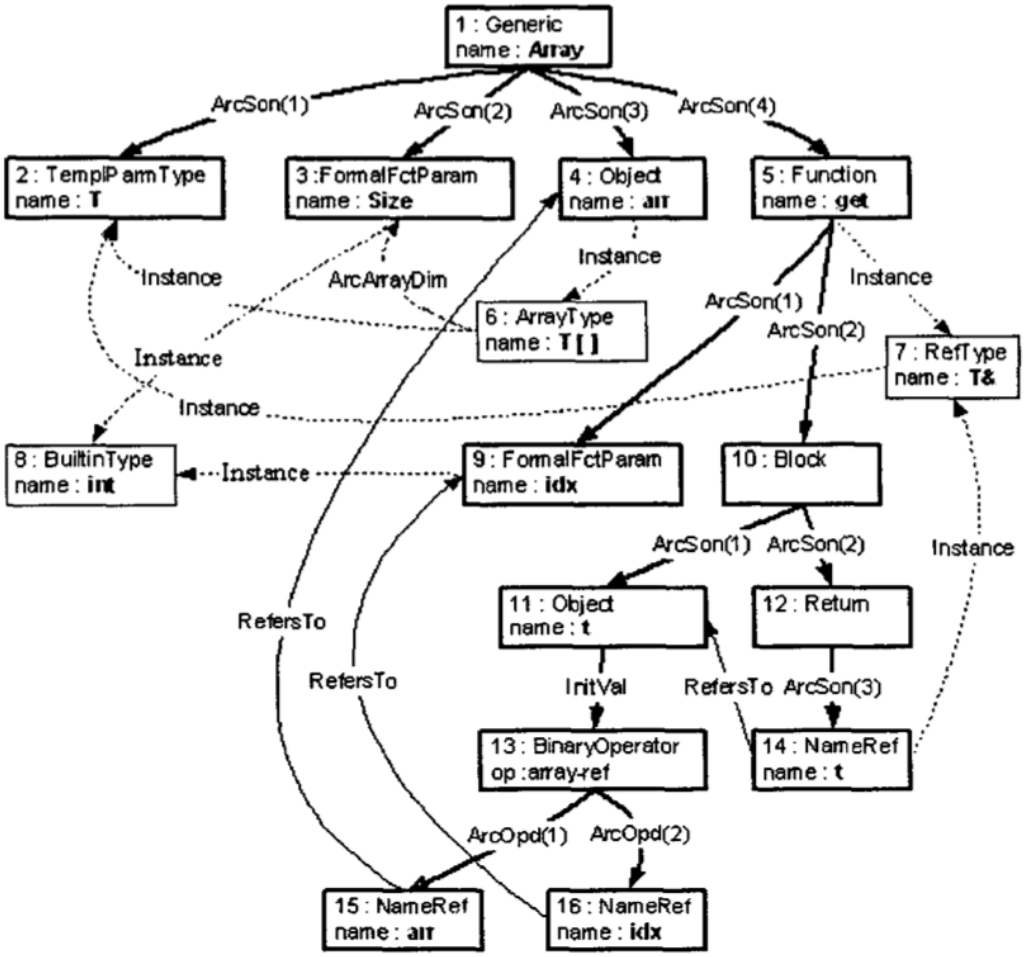
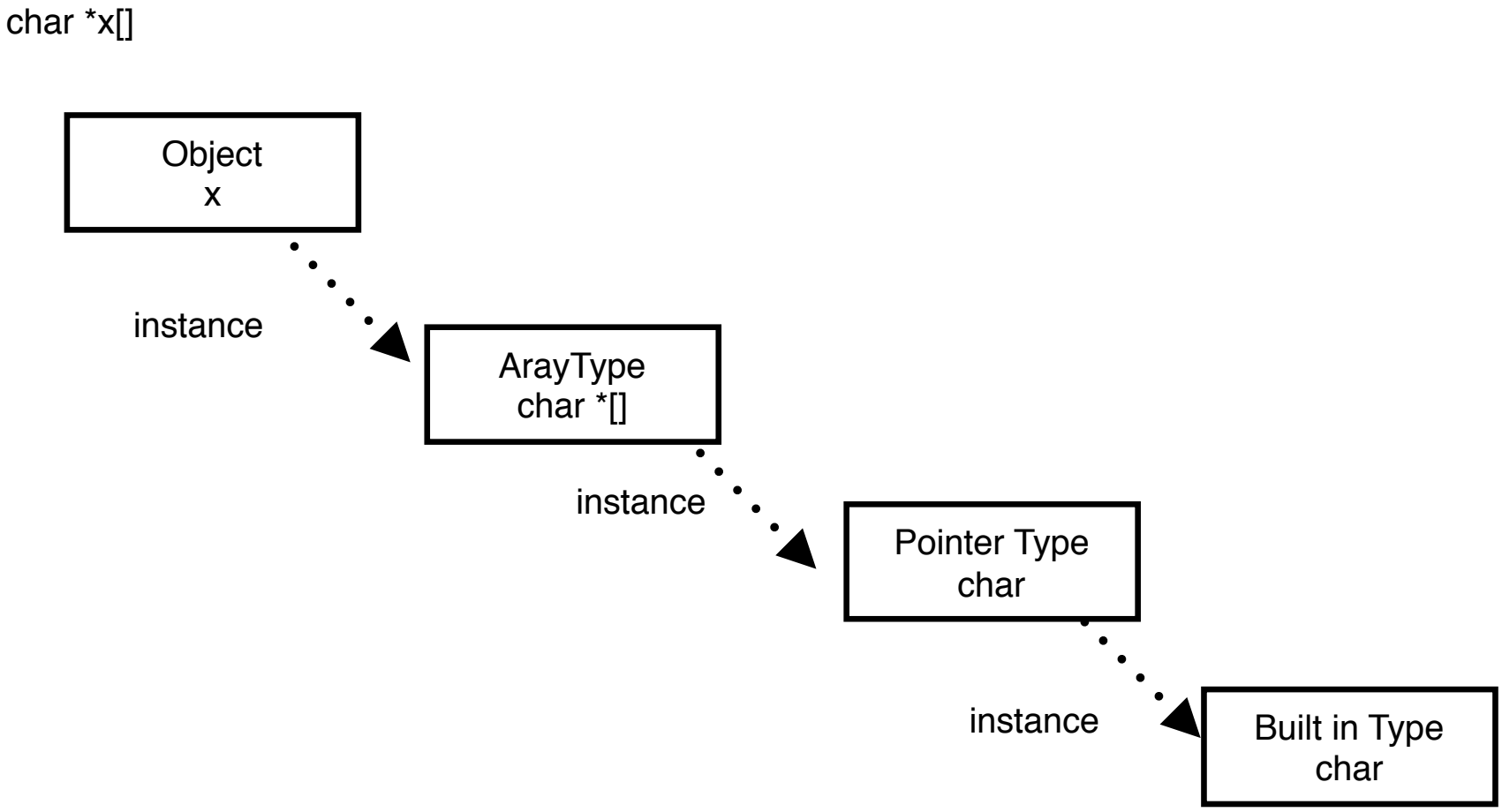
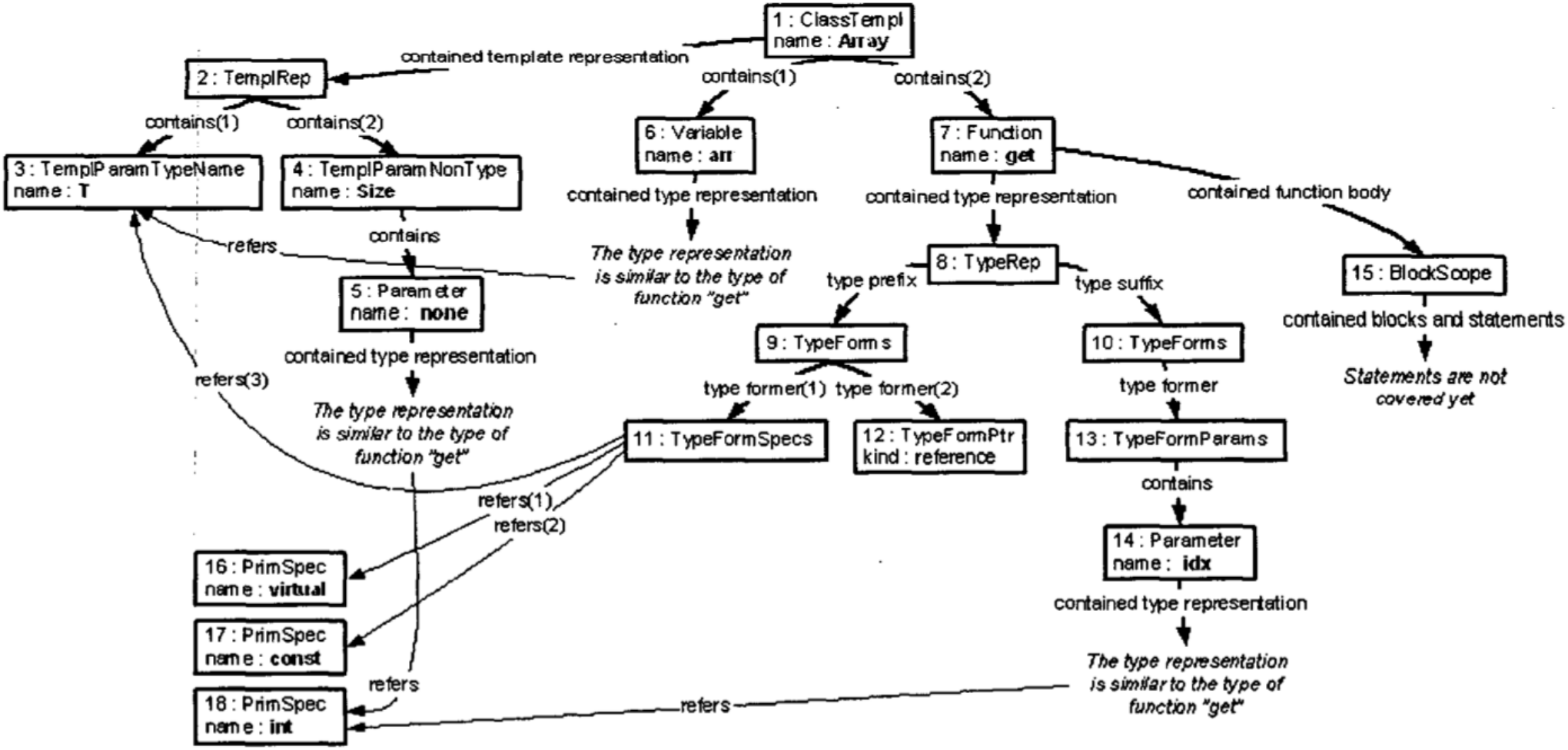


Figure 2. Datrix AST for the common example

Datrix


char *x[]
Object
x
instance
ArayType
char *[]
instance
Pointer Type
char
instance
Built in Type
char

Columbus


1 : ClassTempl
name : Array
contained template representation
2 : TemplRep
contains(1)
contains(2)
6 : Variable
name : arr
7 : Function
name : get
3 : TemplParamTypeName
name : T
4 : TemplParamNonType
name : Size
contained type representation
contained type representation
contained function body
contains
refers
The type representation is similar to the type of function "get"
8 : TypeRep
15 : BlockScope
type prefix
type suffix
5 : Parameter
name : none
9 : TypeForms
10 : TypeForms
contained blocks and statements
contained type representation
type former(1)
type former(2)
type former
Statements are not covered yet
refers(3)
The type representation is similar to the type of function "get"
11 : TypeFormSpecs
12 : TypeFormPtr
kind : reference
13 : TypeFormParams
refers(1)
refers(2)
contains
16 : PrimSpec
name : virtual
14 : Parameter
name : idx
17 : PrimSpec
name : const
contained type representation
refers
The type representation is similar to the type of function "get"
18 : PrimSpec
name : int
refers

Columbus

char * x[]