

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
  int a, b, c, d;
  { ... }

fee() {
  int f[3],g[0],
  h, i, j, k;
  char *p;
  fie(h,i,"ab",j, k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>.\n",
  p,q);
  p = 10;
}
```

What is wrong with this program?
(let me count the ways ...)

CMSC430 Spring 2007

1

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
  int a, b, c, d;
  { ... }

fee() {
  int f[3],g[0],
  h, i, j, k;
  char *p;
  fie(h,i,"ab",j, k);
  k = f * i + j;
  h = g[17];
  printf("<%s,%s>.\n",
  p,q);
  p = 10;
}
```

What is wrong with this program?
(let me count the ways ...)

- declared g[0], used g[17]
- wrong number of args to fie()
- "ab" is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are
"deeper than syntax"

To generate code, we need to understand its meaning !

CMSC430 Spring 2007

2

Beyond Syntax

To generate code, the compiler needs to answer many questions

- Is “x” a scalar, an array, or a function? Is “x” declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of “x” does each use reference?
- Is the expression “x * y + z” type-consistent?
- In “a[i,j,k]”, does a have three dimensions?
- Where can “z” be stored? (*register, local, global, heap, static*)
- In “f ← 15”, how should 15 be represented?
- How many arguments does “fie()” take? What about “printf ()” ?
- Does “*p” reference the result of a “malloc()” ?
- Do “p” & “q” refer to the same memory location?
- Is “x” defined before it is used?

These are beyond a CFG

Beyond Syntax

These questions are part of context-sensitive analysis

- Answers depend on values, not parts of speech
- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
 - > Context-sensitive grammars?
 - > Syntax directed translation using attribute grammars?
(*attributed grammars?*)
- Use *ad-hoc* techniques
 - > Symbol tables
 - > *Ad-hoc* code (*action routines*)

In scanning & parsing, formalism won; different story here.

Beyond Syntax

Telling the story using Syntax Directed Definitions with attribute grammars:

- The attribute grammar formalism is important
 - > Succinctly makes many points clear
 - > Sets the stage for actual, *ad-hoc* practice
- The problems with attribute grammars motivate practice
 - > Non-local computation
 - > Need for centralized information
- Some folks in the community still argue for attribute grammars
 - > Knowledge is power
 - > Information is immunization

We will cover attribute grammars, then move on to *ad-hoc* ideas

Attribute grammars

- Generalization of context-free grammar
- Each grammar symbol has an associated set of attributes
- Augment grammar with rules that define values
- High-level specification, independent of evaluation scheme

Dependences between attributes

- Values are computed from constants & other attributes
- Synthesized attribute - value computed from children
- Inherited attribute - value computed from siblings & parent

Syntax Directed Definitions (SDD)

What is an attribute grammar?

- A context-free grammar augmented with a set of rules
- Each symbol in the derivation has a set of values, or *attributes*
- The rules specify how to compute a value for each attribute

Example grammar

<i>Number</i>	→	<i>Sign List</i>
<i>Sign</i>	→	+
		=
<i>List</i>	→	<i>List Bit</i>
		<i>Bit</i>
<i>Bit</i>	→	0
		1

This grammar describes signed binary numbers

We would like to augment it with rules that compute the decimal value of each valid input string

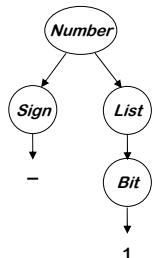
CMSC430 Spring 2007

7

Examples

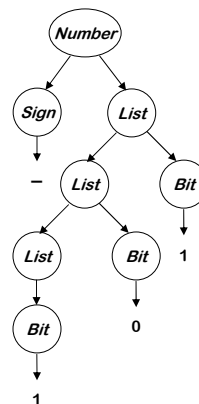
For “-1”

Number → *Sign List*
 → - *List*
 → - *Bit*
 → - 1



For “-101”

Number → *Sign List*
 → *Sign List Bit*
 → *Sign List 1*
 → *Sign List Bit 1*
 → *Sign List 1 1*
 → *Sign Bit 0 1*
 → *Sign 1 0 1*
 → - 101



We will use these two throughout the lecture

CMSC430 Spring 2007

8

Attribute Grammars

Add rules to compute the decimal value of a signed binary number

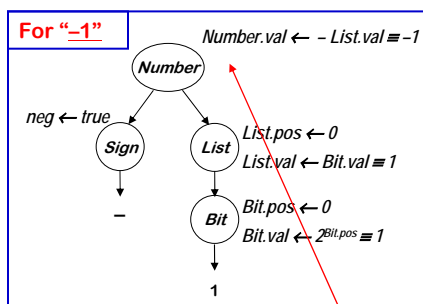
Productions	Attribution Rules
$Number \rightarrow Sign List$	$List.pos \leftarrow 0$ $Number.val \leftarrow$ If $Sign.neg$ then $-List.val$ else $List.val$
$Sign \rightarrow \pm$	$Sign.neg \leftarrow false$
$ =$	$Sign.neg \leftarrow true$
$List_0 \rightarrow List_1 Bit$	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
$ Bit$	$Bit.pos \leftarrow List.pos$ $List.val \leftarrow Bit.val$
$Bit \rightarrow 0$	$Bit.val \leftarrow 0$
$ 1$	$Bit.val \leftarrow 2^{Bit.pos}$

Symbol	Attributes
$Number$	val
$Sign$	neg
$List$	pos, val
Bit	pos, val

CMSC430 Spring 2007

9

Back to the Examples



One possible evaluation order:

- 1 List.pos
- 2 Sign.neg
- 3 Bit.pos
- 4 Bit.val
- 5 List.val
- 6 Number.val

Other orders are possible

Knuth suggested a data-flow model for evaluation

- Independent attributes first
- Others in order as input values become available

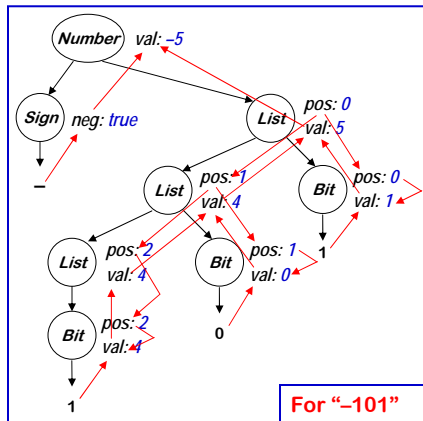
Evaluation order must be consistent with the attribute dependence graph

Rules + parse tree imply an attribute dependence graph

CMSC430 Spring 2007

10

Back to the Examples



This is the complete attribute dependence graph for "-101".

It shows the flow of *all* attribute values in the example.

Some flow downward
→ inherited attributes

Some flow upward
→ synthesized attributes

A rule may use attributes in the parent, children, or siblings of a node

The Rules of the Game

- Attributes associated with nodes in parse tree
- Rules are value assignments associated with productions
- Attribute is defined once, using local information
- Label identical terms in production for uniqueness
- Rules & parse tree define an attribute dependence graph
 - > Graph must be non-circular

This produces a high-level, functional specification

Synthesized attribute

- > Depends on values from children

Inherited attribute

- > Depends on values from siblings & parent

Using Attribute Grammars

Attribute grammars can specify context-sensitive actions

- Take values from syntax
- Perform computations with values
- Insert tests, logic, ...

Synthesized Attributes

- Use values from children & from constants
- S-attributed grammars
- Evaluate in a single bottom-up pass

Good match to LR parsing

Inherited Attributes

- Use values from parent, constants, & siblings
- directly express context
- can rewrite to avoid them
- Thought to be more *natural*

Not easily done at parse time

We want to use both kinds of attribute

Evaluation Methods

Dynamic, dependence-based methods

- Build the parse tree
- Build the dependence graph
- Topological sort the dependence graph
- Define attributes in topological order

Imagine a diminutive person who knows how to evaluate attributed trees ...

Rule-based methods

(treewalk)

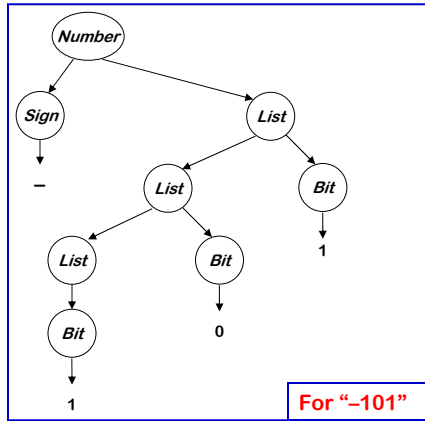
- Analyze rules at compiler-generation time
- Determine a fixed (static) ordering
- Evaluate nodes in that order

Oblivious methods

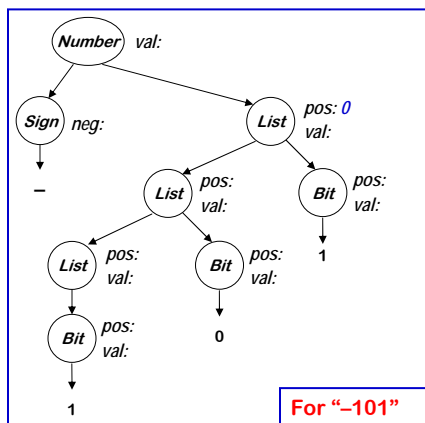
(passes, dataflow)

- Ignore rules & parse tree
- Pick a convenient order (at design time) & use it

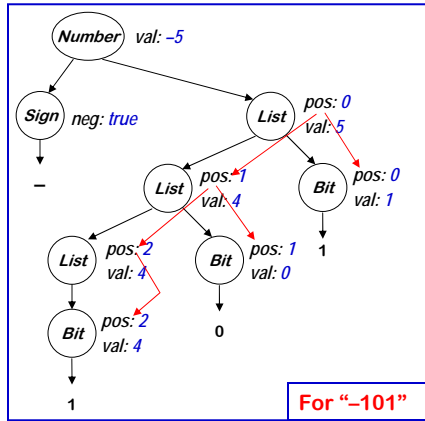
Back to the Example



Back to the Example

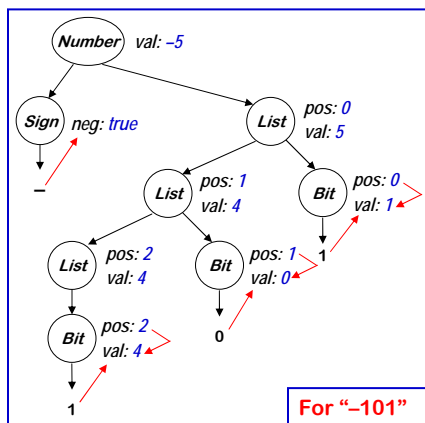


Back to the Example



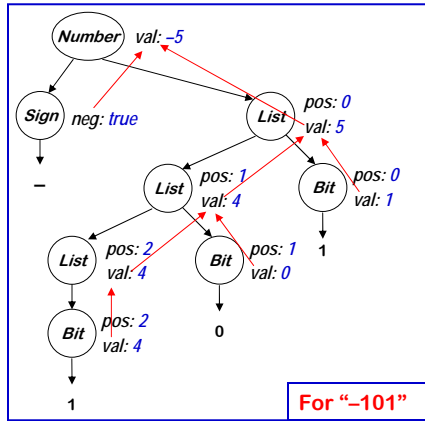
Inherited Attributes

Back to the Example



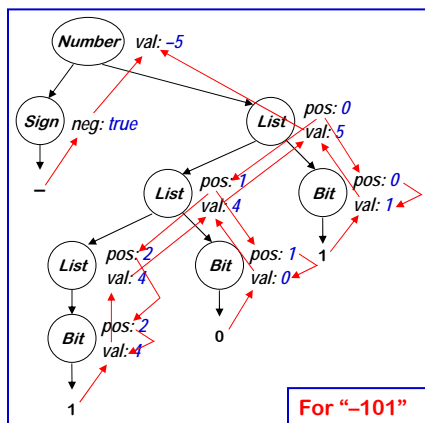
Synthesized attributes

Back to the Example



Synthesized attributes

Back to the Example



If we show the computation ...

& then peel away the parse tree ...

A Circular Attribute Grammar

Productions	Attribution Rules
$Number \rightarrow List$	$List.a \leftarrow 0$
$List_0 \rightarrow List_1 Bit$	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
$\quad \quad Bit$	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
$Bit \rightarrow 0$	$Bit.val \leftarrow 0$
$\quad \quad 1$	$Bit.val \leftarrow 2^{Bit.pos}$

CMSC430 Spring 2007

23

An Extended Example

Grammar for a basic block

(§ 4.3.3)

$Block_0$	\rightarrow	$Block_1 Assign$
	$ $	$Assign$
$Assign$	\rightarrow	$Ident = Expr ;$
$Expr_0$	\rightarrow	$Expr_1 + Term$
	$ $	$Expr_1 - Term$
	$ $	$Term$
$Term_0$	\rightarrow	$Term_1 * Factor$
	$ $	$Term_1 / Factor$
	$ $	$Factor$
$Factor$	\rightarrow	$(Expr)$
	$ $	$Number$
	$ $	$Identifier$

Let's estimate cycle counts

- Each operation has a COST
 - Add them, bottom up
 - Assume a load per value
 - Assume no reuse
- Simple problem for an AG

Hey, this looks useful !

CMSC430 Spring 2007

24

An Extended Example

(continued)

Adding attribution rules

$Block_0$	\rightarrow	$Block_1$ Assign	$Block_0.cost \leftarrow Block_1.cost + Assign.cost$
		Assign	$Block_0.cost \leftarrow Assign.cost$
Assign	\rightarrow	Ident = Expr ;	$Assign.cost \leftarrow COST(store) + Expr.cost$
Expr ₀	\rightarrow	Expr ₁ + Term	$Expr_0.cost \leftarrow Expr_1.cost + COST(add) + Term.cost$
		Expr ₁ - Term	$Expr_0.cost \leftarrow Expr_1.cost + COST(add) + Term.cost$
		Term	$Expr_0.cost \leftarrow Term.cost$
Term ₀	\rightarrow	Term ₁ * Factor	$Term_0.cost \leftarrow Term_1.cost + COST(mult) + Factor.cost$
		Term ₁ / Factor	$Term_0.cost \leftarrow Term_1.cost + COST(div) + Factor.cost$
		Factor	$Term_0.cost \leftarrow Factor.cost$
Factor	\rightarrow	(Expr)	$Factor.cost \leftarrow Expr.cost$
		Number	$Factor.cost \leftarrow COST(load)$
		Identifier	$Factor.cost \leftarrow COST(load)$

All the attributes are synthesized !

CMSC430 Spring 2007

25

An Extended Example

Properties of the example grammar

- All attributes are synthesized \Rightarrow S-attributed grammar
- Rules can be evaluated bottom-up in a single pass
 - > Good fit to bottom-up, shift/reduce parser
- Easily understood solution
- Seems to fit the problem well

What about an improvement?

- Values are loaded only once per block (not at each use)
- Need to track which values have been already loaded

CMSC430 Spring 2007

26

A Better Execution Model

Adding load tracking

- Need sets *Before* and *After* for each production
- Must be initialized, updated, and passed around the tree

Factor	→	(Expr)	Factor.cost ← Expr.cost ; Expr.Before ← Factor.Before ; Factor.After ← Expr.After ; Factor.cost ← COST(load) ; Factor.After ← Factor.Before ; If (Identifier.name ∉ Factor.Before) then Factor.cost ← COST(load) ; Factor.After ← Factor.Before ∪ Identifier.name else Factor.cost ← 0 ; Factor.After ← Factor.Before
		Number	
		Identifier	

This looks more complex!

CMSC430 Spring 2007

27

A Better Execution Model

- Load tracking adds complexity
- But, most of it is in the “copy rules”
- Every production needs rules to copy *Before & After*

A sample production

Expr ₀	→	Expr ₁ + Term	Expr ₀ .cost ← Expr ₁ .cost + COST(add) + Term.cost ; Expr ₁ .Before ← Expr ₀ .Before ; Term.Before ← Expr ₁ .After ; Expr ₀ .After ← Term.After
-------------------	---	--------------------------	--

These copy rules multiply rapidly

Each creates an instance of the set

Lots of work, lots of space, lots of rules to write

CMSC430 Spring 2007

28

An Even Better Model

What about accounting for finite register sets?

- *Before & After* must be of limited size
- Adds complexity to *Factor*→*Identifier*
- Requires more complex initialization

Jump from tracking loads to tracking registers is small

- Copy rules are already in place
- Some local code to perform the allocation

Next

- Curing these problems with *ad-hoc* syntax-directed translation

Remember the previous example?

Grammar for a basic block

<i>Block₀</i>	→	<i>Block₁ Assign</i>
		<i>Assign</i>
<i>Assign</i>	→	<i>Ident = Expr ;</i>
<i>Expr₀</i>	→	<i>Expr₁ + Term</i>
		<i>Expr₁ - Term</i>
		<i>Term</i>
<i>Term₀</i>	→	<i>Term₁ * Factor</i>
		<i>Term₁ / Factor</i>
		<i>Factor</i>
<i>Factor</i>	→	<i>(Expr)</i>
		<i>Number</i>
		<i>Identifier</i>

Let's estimate cycle counts

- Each operation has a COST
- Add them, bottom up
- Assume a load per value
- Assume no reuse

Simple problem for an *AG*

Hey, this looks useful !

And Its Extensions

Tracking loads

- Introduced *Before* and *After* sets to record loads
- Added ≥ 2 copy rules per production
 - > Serialized evaluation into execution order
- Made the whole attribute grammar large & cumbersome

Finite register set

- Complicated one production (*Factor* \rightarrow Identifier)
- Needed a little fancier initialization
- Changes were quite limited

Why is one change hard and the other easy?

The Moral of the Story

- Non-local computation needed lots of supporting rules
- Complex local computation was relatively easy

The Problems

- Copy rules increase cognitive overhead
- Copy rules increase space requirements
 - > Need copies of attributes
 - > Can use pointers, for even more cognitive overhead
- Result is an attributed tree *(somewhat subtle points)*
 - > Must build the parse tree
 - > Must search tree for answers

Addressing the Problem

If you gave this problem to a chief programmer

- Introduce a central repository for facts
- Table of names
 - > Field in table for loaded/not loaded state
- Avoids all the copy rules, allocation & storage headaches
- All inter-assignment attribute flow is through table
 - > Clean, efficient implementation
 - > Good techniques for implementing the table *(hashing)*
 - > When its done, information is in the table ! *(no navigation)*
 - > Cures most of the problems
- Unfortunately, this design violates the functional paradigm
 - > Do we care?

CMSC430 Spring 2007

33

The Realist's Alternative

Ad-hoc syntax-directed translation

- Associate a snippet of code with each production
- At each reduction, the corresponding snippet runs
- Allowing arbitrary code provides complete flexibility
 - > Includes ability to do tasteless & bad things

To make this work

- Need names for attributes of each symbol on *lhs* & *rhs*
 - > Typically, one attribute passed through parser + arbitrary code (structures, globals, statics, ...)
 - > Yacc introduced \$\$, \$1, \$2, ... \$n, left to right
A ::= B C { \$\$ = concat(\$1,\$2) }
- Need an evaluation scheme
 - > Fits nicely into LR(1) parsing algorithm

CMSC430 Spring 2007

34

Reworking the Example (with load tracking)

<i>Block</i> ₀	→ <i>Block</i> ₁ <i>Assign</i> <i>Assign</i>	<i>cost</i> ← 0;
<i>Assign</i>	→ <i>Ident</i> = <i>Expr</i> ;	<i>cost</i> ← <i>cost</i> + <i>COST</i> (store);
<i>Expr</i> ₀	→ <i>Expr</i> ₁ + <i>Term</i> <i>Expr</i> ₁ - <i>Term</i> <i>Term</i>	<i>cost</i> ← <i>cost</i> + <i>COST</i> (add); <i>cost</i> ← <i>cost</i> + <i>COST</i> (sub);
<i>Term</i> ₀	→ <i>Term</i> ₁ * <i>Factor</i> <i>Term</i> ₁ / <i>Factor</i> <i>Factor</i>	<i>cost</i> ← <i>cost</i> + <i>COST</i> (mult); <i>cost</i> ← <i>cost</i> + <i>COST</i> (div);
<i>Factor</i>	→ (<i>Expr</i>) Number Identifier	<i>cost</i> ← <i>cost</i> + <i>COST</i> (loadi); { <i>i</i> ← hash(Identifier); if (<i>Table</i> [<i>i</i>].loaded = false) then { <i>cost</i> ← <i>cost</i> + <i>COST</i> (load); <i>Table</i> [<i>i</i>].loaded ← true; } }

This looks cleaner & simpler !

Example — Building a Parse Tree

- Assume constructors for each node
- Assume stack holds pointers to nodes
- Assume yacc syntax

<i>Goal</i>	→ <i>Expr</i>	\$\$ = \$1;
<i>Expr</i>	→ <i>Expr</i> + <i>Term</i> <i>Expr</i> - <i>Term</i> <i>Term</i>	\$\$ = MakeAddNode(\$1,\$3); \$\$ = MakeSubNode(\$1,\$3); \$\$ = \$1;
<i>Term</i>	→ <i>Term</i> * <i>Factor</i> <i>Term</i> / <i>Factor</i> <i>Factor</i>	\$\$ = MakeMulNode(\$1,\$3); \$\$ = MakeDivNode(\$1,\$3); \$\$ = \$1;
<i>Factor</i>	→ (<i>Expr</i>) <u>number</u> <u>id</u>	\$\$ = \$1; \$\$ = MakeNumNode(token); \$\$ = MakeIdNode(token);

Reality

Most parsers are based on this *ad-hoc* style of context-sensitive analysis

Advantages

- Addresses the shortcomings of the AG paradigm
- Efficient, flexible

Disadvantages

- Must write the code with little assistance
- Programmer deals directly with the details

Most parser generators support a yacc-like notation

Typical Uses

- Building a symbol table
 - > Enter declaration information as processed
 - > At end of declaration syntax, do some post processing
 - > Use table to check errors as parsing progresses
 - **Note:** this assumes table is **global**
- Simple error checking/type checking
 - > Define before use → lookup on reference
 - > Dimension, type, ... → check as encountered
 - > Type conformability of expression → bottom-up walk
 - > Procedure interfaces are harder
 - Build a representation for parameter list & types
 - Create list of sites to check
 - Either check offline, or handle the cases for arbitrary orderings

Is This Really “Ad-hoc” ?

Relationship between practice and attribute grammars

Similarities

- Both rules & actions associated with productions
- Application order determined by tools, not author
- (Somewhat) abstract names for symbols

Differences

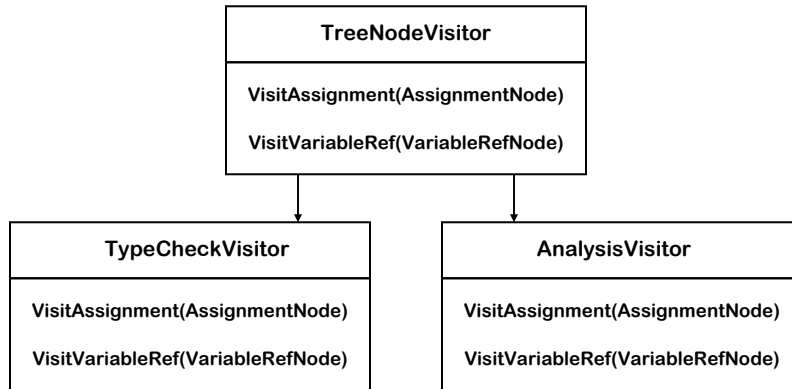
- Actions applied as a unit; not true for AG rules
- Anything goes in *ad-hoc* actions; AG rules are functional
- AG rules are higher level than *ad-hoc* actions

Limitations

- Forced to evaluate in a given order: *postorder*
 - > Left to right only
 - > Bottom up only
 - Implications
 - > Declarations before uses
 - > Context information cannot be passed down
 - How do you know what rule you are called from within?
 - Example: cannot pass bit position from right down
- List → List Bit : $$$ = 2 * \$1 + \$2$
- > Could you use globals?
 - In this case we could get the position from the left, which is not much help (and it requires initialization)

Alternative Strategy

- Build Abstract Syntax Tree
 - > Use tree walk routines
 - > Use “visitor” design pattern to add functionality

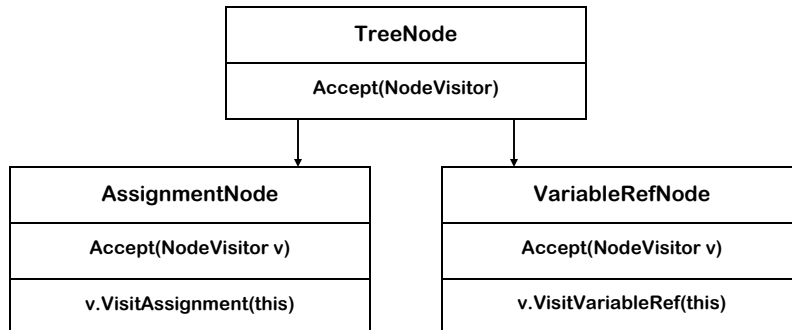


CMSC430 Spring 2007

41

Visitor Treewalk I

- Parallel structure of tree:
 - > Separates treewalk code from node handling code
 - > Facilitates processing change without change to tree structure




CMSC430 Spring 2007

42

Visitor Treewalk II

```
VisitAssignment(aNodePtr)
    // preprocess assignment
    (aNodePtr->rhs)->Accept(this);
    // postprocess rhs info;
    (aNodePtr->lhs)->Accept(this);
    // postprocess assignment;
```

 Refers to current visitor!

To start the process:

```
AnalysisVisitor a; treeRoot->Accept(a);
```

Summary: Strategies for Context-Sensitive Analysis

- **Attribute Grammars**
 - > **Pros:** Formal, powerful, can deal with propagation strategies
 - > **Cons:** Too many copy rules, no global tables, works on parse tree
- **Postorder Code Execution**
 - > **Pros:** Simple and functional, can be specified in grammar (Yacc) but does not require parse tree
 - > **Cons:** Rigid evaluation order, no context inheritance
- **Generalized Tree Walk**
 - > **Pros:** Full power and generality, operates on abstract syntax tree (using Visitor pattern)
 - > **Cons:** Requires specific code for each tree node type, more complicated

Type systems

Types

- Values that share a set of common properties defined by language and/or programmer

Type system

1. set of types in a programming language, and
2. rules that use types to specify program behavior

Example type rules

- If operands of addition are of type integer, then result is of type integer
- The result of the unary & operator is a pointer to the object referred to by the operand

Advantages of typed languages

- Ensure run-time safety
- Expressiveness (overloading, polymorphism)
- Provide information for code generation

CMSC430 Spring 2007

45

Type checking

Type checker

- Enforces rules of type system
- May be strong/weak, static/dynamic

Static type checking

- Performed at compile time
- Early detection, no run-time overhead
- Not always possible (e.g., $A[i]$)

Dynamic type checking

- Performed at run time
- More flexible, rapid prototyping
- Overhead to check run-time type tags

CMSC430 Spring 2007

46

Type expressions

Type expressions

- Used to represent the type of a language construct
- Describes both language and programmer types

Examples

- Basic types: integer, real, character, ...
- Constructed types: arrays, records, pointers, functions,...

Constructing new types

- arrays *array(1..10, T)*
- records *T1 x T2 ...*
- pointers *pointer(T)*
- functions *T1 x T2 x ... → T_n*

CMSC430 Spring 2007

47

A simple type checker

Using a synthesized attribute grammar, we will describe a type checker for arrays, pointers, statements, and functions.

Grammar for source language:

$P ::= L ; E$

$L ::= L ; D \mid D$

$D ::= D ; E \mid \text{id} : T$

$T ::= \text{char} \mid \text{integer} \mid \text{array} [\text{num}] \text{ of } T \mid \uparrow T$

$E ::= \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E[E] \mid E \uparrow$

Basic types *char, integer, typeError*

assume all arrays start at 1, e.g., *array [256] of char* results in the type expression *array(1..256,char)*

- \uparrow builds a pointer type, so $\uparrow \text{integer}$ results in the type expression *pointer(integer)*

CMSC430 Spring 2007

48

Type checking example

Partial attribute grammar for the type system

$L ::= L D$

$L ::= D$

$D ::= \text{id}: T \quad \{ \text{addtype}(\text{id.entry}, T.\text{type}) \}$

$T ::= \text{char} \quad \{ T.\text{type} \leftarrow \text{char} \}$

$T ::= \text{integer} \quad \{ T.\text{type} \leftarrow \text{integer} \}$

$T ::= \uparrow T_1 \quad \{ T.\text{type} \leftarrow \text{pointer}(T_1.\text{type}) \}$

$T ::= \text{array} [\text{num}] \text{ of } T_1 \quad \{ T.\text{type} \leftarrow \text{array}(1: \dots \text{num.val}, T_1.\text{type}) \}$

Type checking expressions

Each expression is assigned a type using rules associated with the grammar.

- $E ::= \text{literal} \quad \{ E.\text{type} \leftarrow \text{char} \}$
- $E ::= \text{num} \quad \{ E.\text{type} \leftarrow \text{integer} \}$
- $E ::= \text{id} \quad \{ E.\text{type} \leftarrow \text{lookup}(\text{id.entry}) \}$
- $E ::= E_1 \text{ mod } E_2 \quad \{ E.\text{type} \leftarrow \text{if } E_1.\text{type} = \text{integer and } E_2.\text{type} = \text{integer then integer else typeError} \}$
- $E ::= E_1[E_2] \quad \{ E.\text{type} \leftarrow \text{if } E_2.\text{type} = \text{integer and } E_1.\text{type} = \text{array}(s,t) \text{ then } t \text{ else typeError} \}$
- $E ::= E_1 \uparrow \quad \{ E.\text{type} \leftarrow \text{if } E_1.\text{type} = \text{pointer then } t \text{ such that } \text{pointer}(E_1, t) \text{ else typeError} \}$

Type checking statements

Statements do not typically have values, therefore we assign them the type void. If an error is detected within the statement, it gets type `typeError`.

$S ::= id \leftarrow E \quad \{ S.type \leftarrow \text{if } id.type = E.type \text{ then void else } typeError \}$

$S ::= \text{if } E \text{ then } S_1 \quad \{ S.type \leftarrow \text{if } E.type = \text{boolean} \text{ then } S_1.type \text{ else } typeError \}$

$S ::= \text{while } E \text{ do } S_1 \quad \{ S.type \leftarrow \text{if } E.type = \text{boolean} \text{ then } S_1.type \text{ else } typeError \}$

$S ::= S_1 ; S_2 \quad \{ S.type \leftarrow \text{if } S_1.type = \text{void} \text{ then } S_2.type \text{ else } typeError \}$

Type checking functions

We add two new productions to the grammar to represent function declarations and applications

$T ::= T \rightarrow T \quad \text{declaration}$

$E ::= E (E) \quad \text{application}$

To capture the argument and return type, we use

$T ::= T_1 \rightarrow T_2 \quad \{ T.type \leftarrow (T_1.type \rightarrow T_2.type) \}$

$E ::= E_1 (E_2) \quad \{ E.type \leftarrow \text{if } E_1.type = s \rightarrow t \text{ and } E_2.type = s \text{ then } t \text{ else } typeError \}$

Note: We could avoid functions *addtype* and *lookup* by adding a new inherited attribute *symboltable* which was a set that kept track of declarations, and was passed down parse tree. (How?)