UNIVERSITY OF TWENTE.

COMPILER CONSTRUCTION:

CC 3-2: ATTRIBUTE GRAMMARS

MODULE 8: PROGRAMMING PARADIGMS

8 MAY 2019

PREVIOUSLY:

TYPE CHECKING AND INFERENCE

- In parse tree, every node is (an instance of) a grammar symbol
  - Symbol is terminal (= token type) or nonterminal
- Some symbols are typed
  - Namely, those corresponding to expressions
- How can the type of the parse tree node be established?
  1. *Synthesized* from symbol & children in parse tree
     - E.g. type of x+2 in Java?

       int if x is **int**, String if x is String, invalid if x is Object
  2. *Inherited* from siblings & parent in parse tree
     - E.g., type of x, y, z in **if** (x & y) { z = 3; }?

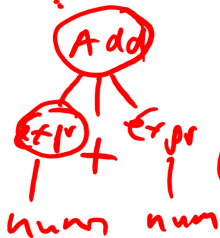
       x, y are **boolean** because x & y is **boolean** (which is what **if** expects),
       z is **int** because z = 3 is **int** (which is because 3 is **int**, synthesized from 3)
     - Especially important in implicitly typed languages
- Type errors:
  - Type synthesis: error if subnodes of symbol have “wrong” type
  - Type inheritance: error if inherited type differs from actual (synthesized) type

PREVIOUSLY:

TYPING RULES

- *Not* part of the actual grammar
  - Context-free grammars not powerful enough
  - Could be done using context-sensitive grammars, but not practical
- Rules added on top of grammar
  - Attribute rules → “attributed grammar”
  - Kinds of rules: Synthesized versus inherited
  - Not just used for typing
- In Antlr, *tree listeners* (or *visitors*) are used to implement such rules

ATTRIBUTES AND ATTRIBUTE RULES



- Central notion: parse tree
- Every parse tree node has a *type*, namely a combination of:
  1. The associated grammar symbol
     - Token type or non-terminal
  2. The grammar rule that produced it
     - For non-leaf nodes

> We're not talking about expression types now!

- *Attribute*: some value associated with a parse tree node
  - Set of attributes of a node depends on the node type (kind 1 above)
- *Attribute rule*: how to compute attributes from neighbouring nodes
  - From parent or sibling nodes: *inherited* attribute
  - From children or self: *synthesized* attribute
- Attribute rules are defined per individual grammar rule
  - Express the computation for any parse node produced by that rule (kind 2)
  - One (attribute) rule for each attribute of that parse tree node type (kind 1)

ATTRIBUTES: EXAMPLE (EC FIG. 4.4)

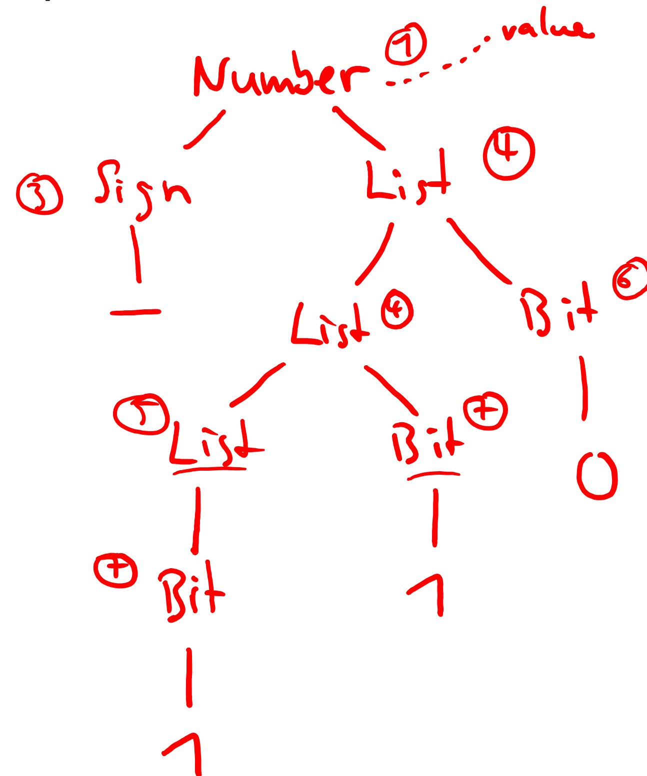
Kind 1 type

Kind 2 type

- Signed binary numbers

1.	*Number*	→	*Sign List*
2.	*Sign*	→	'+'
3.		|	'–'
4.	$List_0$	→	$List_1$ *Bit*
5.		|	*Bit*
6.	*Bit*	→	'0'
7.		|	'1'

-110 parses to



Attributes

Number:	value	(int)
Sign:	isNeg	(bool)
List:	value	(int)
Bit:	isZero	(bool)

Attribute rules:

(6) Bit.isZero := true

(7) Bit.isZero := false

(5) $List_0$.value := if Bit.isZero then 0 else 1

(4) $List_0$.value := (if Bit.isZero then 0 else 1) + $List_1$.value * 2

ATTRIBUTES: EXAMPLE (EC FIG. 4.4)

Kind 1 type

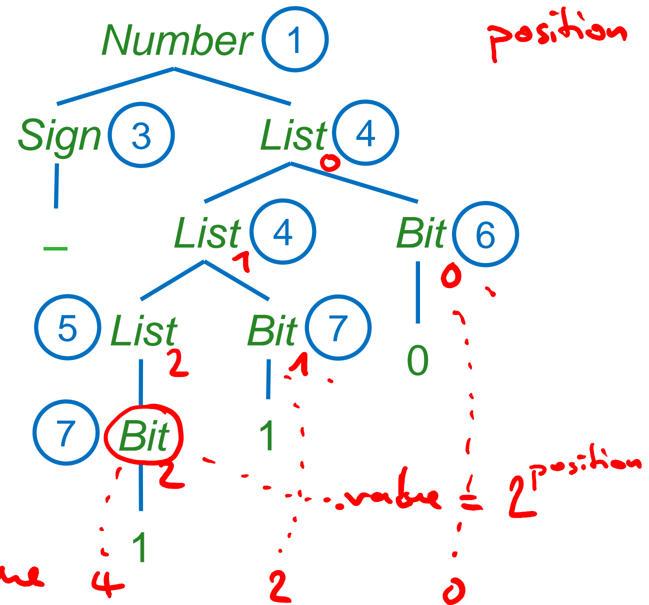
Kind 2 type

- Signed binary numbers

1.	*Number*	→	*Sign List*
2.	*Sign*	→	'+'
3.		|	'–'
4.	*List*	→	*List Bit*
5.		|	*Bit*
6.	*Bit*	→	'0'
7.		|	'1'

-110 parses to

Kind 2 types?



- Suppose we want to compute the value of a *Number* as an attribute
  - What auxiliary attributes do we need?
    - *Number* **int** value
    - *Sign* **boolean** negative
    - *List* **int** value, **int** position (position is distance of last bit to end)
    - *Bit* **int** value, **int** position

There are alternative solutions

HOW TO COMPUTE THE ATTRIBUTES?

For *Number*: **int** value
For *Sign*: **boolean** negative
For *List*: **int** value, **int** position
For *Bit* : **int** value, **int** position

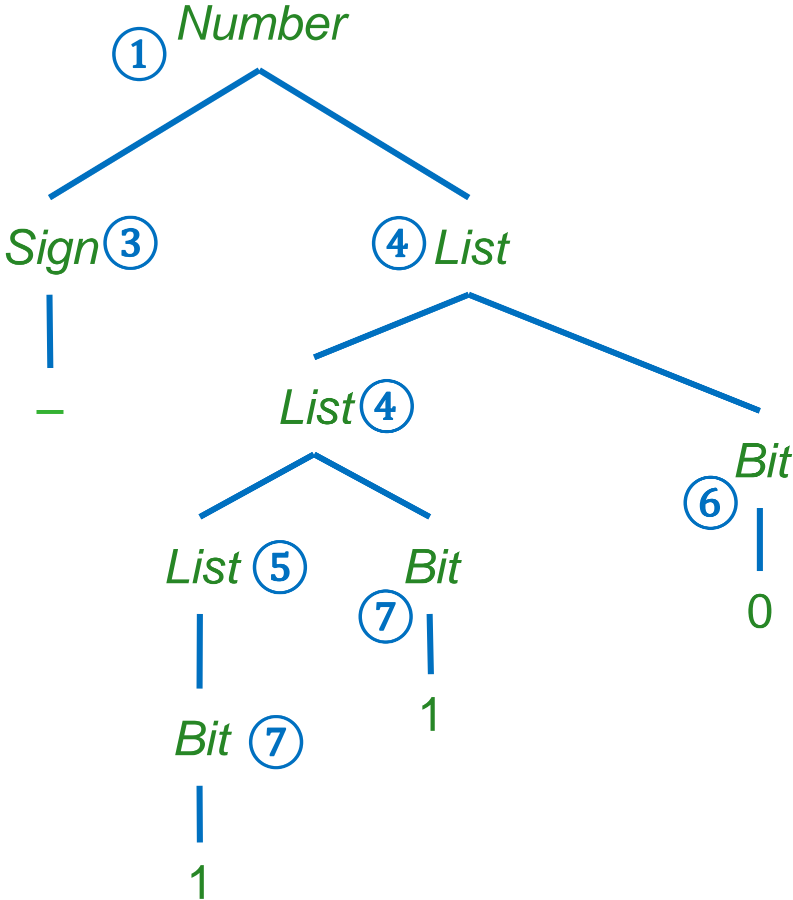
Signed binary numbers

1.	*Number* → *Sign* *List*		*List*.position ← 0
			Number.value ← *Sign*.negative ? –*List*.value : *List*.value
2.	*Sign*	→ '+'	*Sign*.negative ← false
3.		| '–'	*Sign*.negative ← true
4.	$List_0$	→ $List_1$ *Bit*	$List_1$.position ← $List_0$.position + 1
			Bit.position ← $List_0$.position
			$List_0$.value ← $List_1$.value + *Bit*.value
5.	*List*	→ *Bit*	*Bit*.position ← *List*.position
			List.value ← *Bit*.value
6.	*Bit*	→ '0'	*Bit*.value ← 0
7.		| '1'	*Bit*.value ← $2^{Bit.position}$

Note:

- *List*.position and *Bit*.position are passed from node to children (inherited)
- *Number*.value, *Bit*.value and *Sign*.negative from children to node (synthesized)

EXAMPLE COMPUTATION



① *List*.p ← 0
  Number.v ← *Sign*.n ? –*List*.v : *List*.v

② *Sign*.n ← false

③ *Sign*.n ← true

④ $List_1$.p ← $List_0$.p + 1
  Bit.p ← $List_0$.p
  $List_0$.v ← $List_1$.v + *Bit*.v

⑤ *Bit*.p ← *List*.p
  List.v ← *Bit*.v

⑥ *Bit*.v ← 0

⑦ *Bit*.v ← $2^{Bit.p}$

EXAMPLE COMPUTATION

① *List*.p ← 0
Number.v ← *Sign*.n ? –*List*.v : *List*.v

② *Sign*.n ← false

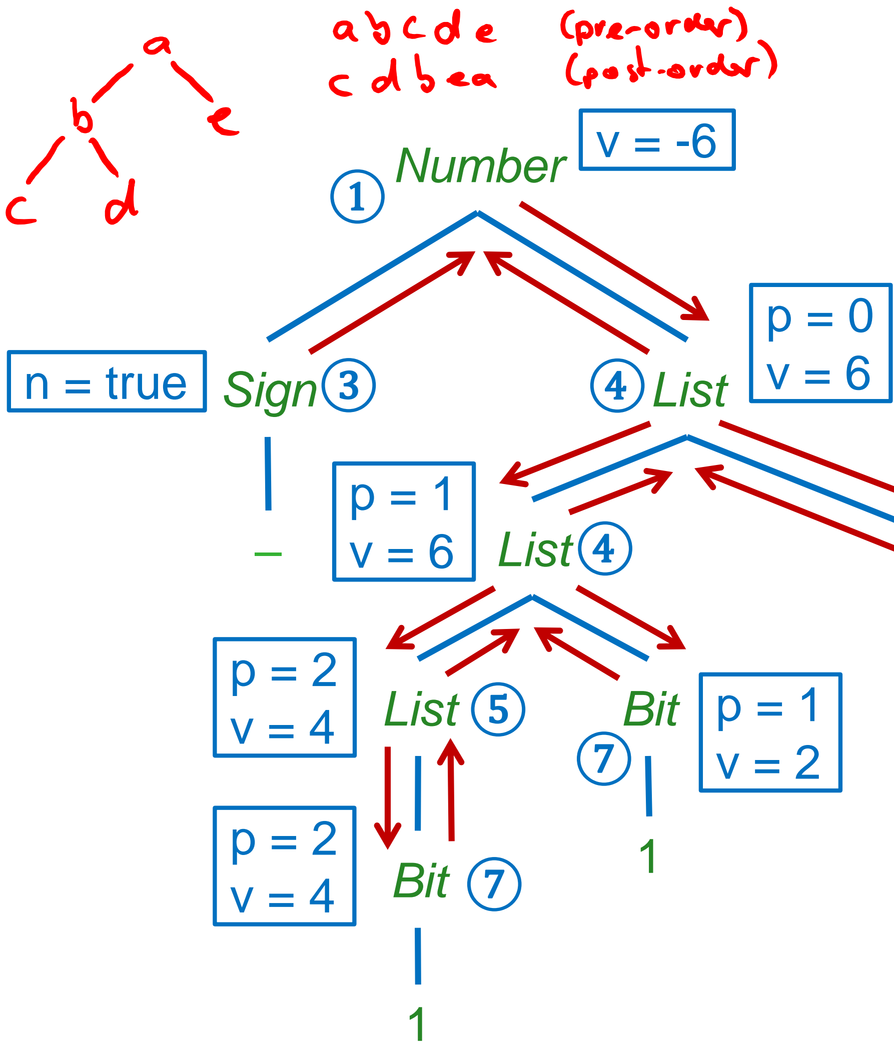
③ *Sign*.n ← true

④ $List_1$.p ← $List_0$.p + 1
Bit.p ← $List_0$.p
$List_0$.v ← $List_1$.v + *Bit*.v

⑤ *Bit*.p ← *List*.p
List.v ← *Bit*.v

⑥ *Bit*.v ← 0

⑦ *Bit*.v ← $2^{Bit.p}$



- Attribute rules are declarative
- Downward rules: *inherited*
- Upward rules: *synthesized*
- No explicit order of traversal

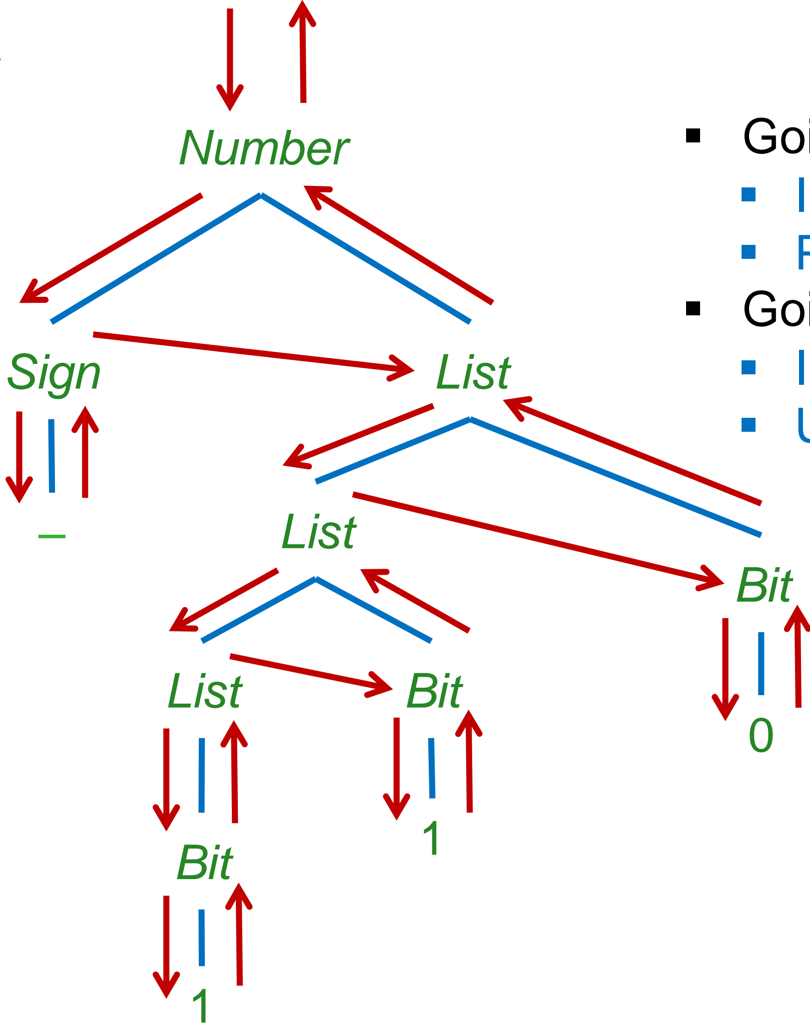
THE ANTLR WAY OF DOING THINGS

- Use instance of `MyGrammarListener`
  - By extending `MyGrammarBaseListener`
- This visits the tree in preorder *and* postorder
  - Preorder visiting methods: `enterNonterminal`
  - Postorder visiting methods: `exitNonterminal`
  - Terminal visiting methods: `visitTerminal`
  - In addition: `enterEveryRule`, `exitEveryRule`
- Attributes: through instances of `ParseTreeProperty<AttrType>`
  - This is essentially a `Map<ParseTree,AttrType>` (you may use that instead)
  - E.g., for Signed Binary Numbers:

```
ParseTreeProperty<Integer> value; // keys: Number/List/Bit nodes
ParseTreeProperty<Boolean> negative; // keys: Sign nodes
ParseTreeProperty<Integer> position; // keys: List/Bit nodes
```

- Previous versions of Antlr: parse tree nodes extended with attribute fields

COMBINED PREORDER AND POSTORDER TRAVERSAL



- Going down: preorder (`enter...`)
  - Implements *inheritance*
  - Prepare stuff for your children
- Going up: postorder (`exit...`)
  - Implements *synthesis*
  - Use stuff of your children

THE ANTLR WAY OF DOING THINGS: RULE TYPES

- How to distinguish between "rule types, kind 2"?
  1. *List* → *List Bit*
  2. | *Bit*
- Give *tags* to top-level rule alternatives

```
list : list bit   #longList
     | bit        #shortList
     ;
```

Your names for kind 2 rule types

  - Now the listener methods are not called `enterList`, `exitList` but `enterLongList`, `exitLongList`, `enterShortList`, `exitShortList`

THE ANTLR WAY OF DOING THINGS: USING THE RHS

- How to get to symbols in the RHS? E.g., $List_1$ and Bit in
  - $List_0 \rightarrow List_1 \ Bit$ $\quad$ $List_1$.position ← $List_0$.position + 1
    Bit.position ← $List_0$.position
    $List_0$.value ← $List_1$.value + Bit.value
- Use *context* parameters of `enter`- and `exit`-methods
- `list : list bit #longList`
  - results in `public void enterLongList(LongListContext ctx)`
  - `ctx` stands for the current parse tree node; `LongListContext` has methods
    - ctx.`list()` returning the first child of this parse tree node
    - `bit()` returning the second child of this parse tree node
- `expr: expr PLUS expr #plusExpr`
  - results in `public void enterPlusExpr(PlusExprContext ctx)`
  - `PlusExprContext` has methods
    - `expr()` returning the *list* of (two) `expr`-children
    - `PLUS()` returning the `PLUS`-child

ctx.expr(0)

ctx.expr(1)

- or name the nonterminals:
  expr : leftExpr = expr PLUS rightExpr = expr # longList
  | ...

THE ANTLR WAY OF DOING THINGS: USING THE RHS

- How to get to symbols in the RHS? E.g., $List_1$ and *Bit* in
  - $List_0 \rightarrow List_1 \ \ Bit$ $\quad$ $List_1.\text{position} \leftarrow List_0.\text{position} + 1$
$Bit.\text{position} \leftarrow List_0.\text{position}$
$List_0.\text{value} \leftarrow List_1.\text{value} + Bit.\text{value}$
- Use *context* parameters of `enter`- and `exit`-methods
- `list : list bit  #longList`
  - results in `public void enterLongList(LongListContext ctx)`
  - `ctx` stands for the current parse tree node; `LongListContext` has methods
    - `list()` returning the first child of this parse tree node
    - `bit()` returning the second child of this parse tree node
- `expr: expr PLUS expr  #plusExpr`
  - results in `public void enterPlusExpr(PlusExprContext ctx)`
  - `PlusExprContext` has methods
    - `expr()` returning the *list* of (two) `expr`-children
    - `PLUS()` returning the `PLUS`-child
- You can also use `ctx.getChild(index)`, but why should you?

THE ANTLR WAY: USING PARSE TREE PROPERTIES

$List_0 \rightarrow List_1 \; Bit \qquad List_1.\text{position} \leftarrow List_0.\text{position} + 1$
$\qquad\qquad\qquad\qquad Bit.\text{position} \leftarrow List_0.\text{position}$
$\qquad\qquad\qquad\qquad List_0.\text{value} \leftarrow List_1.\text{value} + Bit.\text{value}$

- How to program this combined inherited/synthesised attribute rule?
  - Declare instance variables

```
ParseTreeProperty<Integer> value; // keys: Number/List/Bit nodes
ParseTreeProperty<Boolean> negative; // keys: Sign nodes
ParseTreeProperty<Integer> position; // keys: List/Bit nodes
```

  - Inherited attributes (`enter`…): $List_1$ $List_0$

```
public void enterLongList(LongListContext ctx) {
    position.put(ctx.list(), position.get(ctx)+1);
    position.put(ctx.bit(), position.get(ctx));
}
```

  - Synthesized attributes (`exit`…):

```
public void exitLongList(LongListContext ctx) {
    value.put(ctx, value.get(ctx.list())+value.get(ctx.bit()));
}
```