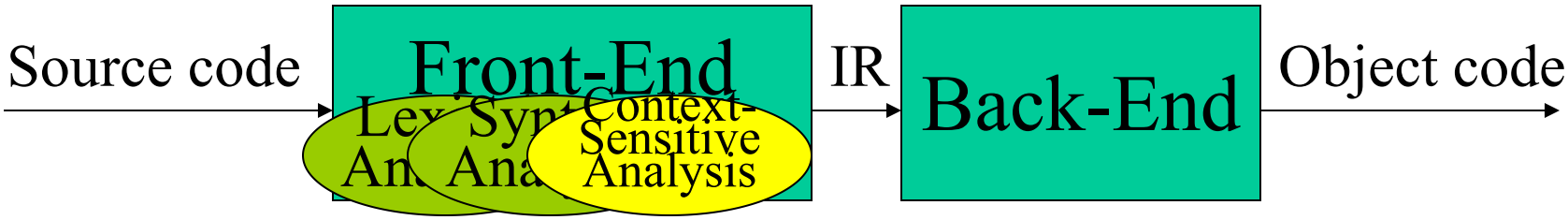
Lecture 11: Context Sensitive Analysis



(from last lectures) :

- After Lexical & Syntax analysis we have determined that the input program is a valid sentence in the source language.
- The outcome from syntax analysis is a parse tree: a graphical representation of how the start symbol of a grammar derives a string in the language.
- *But, have we finished with analysis & detection of all possible errors?*
- Today's lecture: attribute grammars

What is wrong with the following?

```
foo(a,b,c,d)
  int a,b,c,d;
{ … }

lala()
{
  int f[3],g[0],h,i,j,k;
  char *p;
  foo(h,i,"ab",j,k);
  k=f*i+j;
  h=g[7];
  printf("%s,%s",p,q);
  p=10;
}
```

I can count (at least) 6 errors!

All of which are beyond syntax!

These are not issues for the ***context-free*** grammar!

To generate code, we need to understand its meaning!

Beyond syntax: context sensitive questions

To generate code, the compiler needs to answer questions such as:

- 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-2*y`** type-consistent? (**type checking**: the compiler needs to assign a type to each expression it calculates)
- In **`a[i,j,k]`**, is **`a`** declared to have 3 dimensions?
- Who is responsible to allocate space for **`z`**? For how long its value needs to be preserved?
- How do we represent **`15`** in **`f=15`**?
- How many arguments does **`foo()`** take?
- Can **`p`** and **`q`** refer to the same memory location?

These are beyond a context-free grammar!

Type Systems

- A value's **type** is the set of properties associated with it. The set of types in a programming language is called **type system**. **Type checking** refers to assigning/inferring types for expressions and checking that they are used in contexts that are legal.
- Components of a type system:
  - Base or Built-in Types (e.g., numbers, characters, booleans)
  - Rules to: construct new types; determine if two types are equivalent; infer the type of source language expressions.
- Type checking: (depends on the design of the language and the implementation)
  - At compile-time: statically checked
  - At run-time: dynamically checked

+	**int**	**real**	**double**	**complex**
int	int	real	double	complex
real	real	real	double	complex
double	double	double	double	illegal
complex	complex	complex	illegal	complex

- If the language has a 'declare before use' policy, then inference is not difficult. The real challenge is when no declarations are needed.

Context-sensitive questions

The questions in Slide 3 are part of context-sensitive analysis:

- answers depend on values, not syntax.
- questions and answers involve non-local information.
- answers may involve computation.

How can we answer these questions?

- Use formal methods:
  - use context-sensitive grammars: many important questions would be difficult to encode in a Context-Sensitive Grammar and the set of rules would be too large to be manageable (e.g., the issue of declaration before use).
  - attribute grammars.
- Use *ad hoc* techniques:
  - Symbol tables and *ad hoc*, syntax-directed translation using attribute grammars.

In scanning and parsing formalism won; here it is a different story!

Attribute Grammars

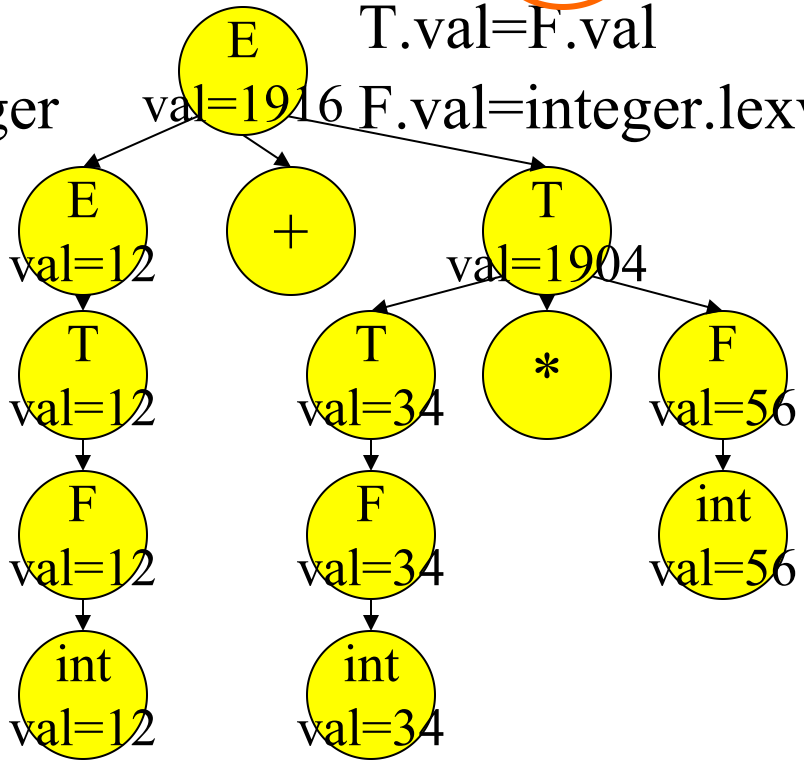
- Idea:
  - annotate each grammar symbol with a set of values or attributes.
  - associate a semantic rule with each production rule that defines the value of each attribute in terms of other attributes.
- Attribute Grammars (a formalisation by Knuth):
  - A context-free grammar augmented with a set of (semantic) rules.
  - Each symbol has a set of values (or attributes).
  - The rules specify how to compute a value for each attribute.

Example

(semantic rules to calculate the value of an expression)

$G \rightarrow E$	print(E.val)
$E \rightarrow E_1+T$	$E.val=E_1.val+T.val$
$\mid T$	E.val=T.val
$T \rightarrow T_1*F$	$T.val=T_1.val \times F.val$
$\mid F$	T.val=F.val
$F \rightarrow integer$	F.val=integer.lexval

NB: we label identical terms uniquely!



Evaluation order:
start from the leaves
and proceed bottom-up!

Attributes

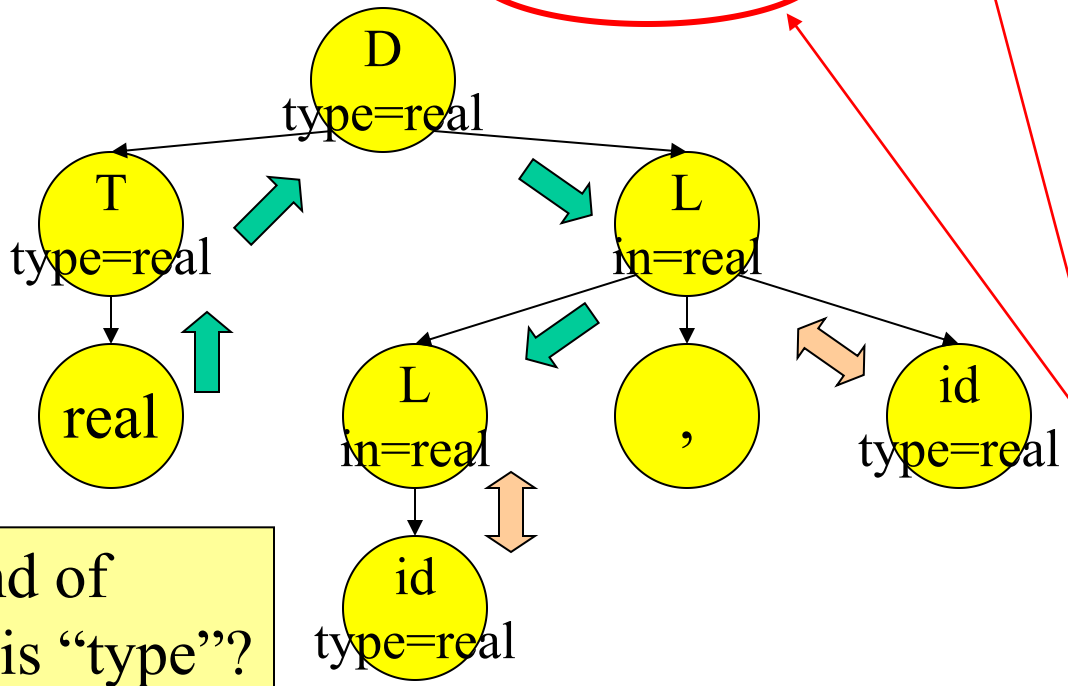
Dependences between attributes:

- Synthesised attributes: derive their value from constants and children.
  - Only synthesised attributes: S-attribute grammars (can be evaluated in one bottom-up pass; cf. with the example before).
- Inherited attributes: derive their value from parent, constants and siblings.
  - Directly express context.

***Issue**: semantic rules of one node of the parse tree define dependencies with semantic rules of other nodes. What about the evaluation order in more complex cases? (we'll come to this later…)*

Example (variable declarations) cf. Aho1, p.283, Ex: 5.3

$D \rightarrow T\ L$	L.in=T.type
$T \rightarrow$ int	T.type=integer
| real	T.type=real
$L \rightarrow L_1$, id	L_1.in=L.in; id.type=L.in
| id	id.type=L.in

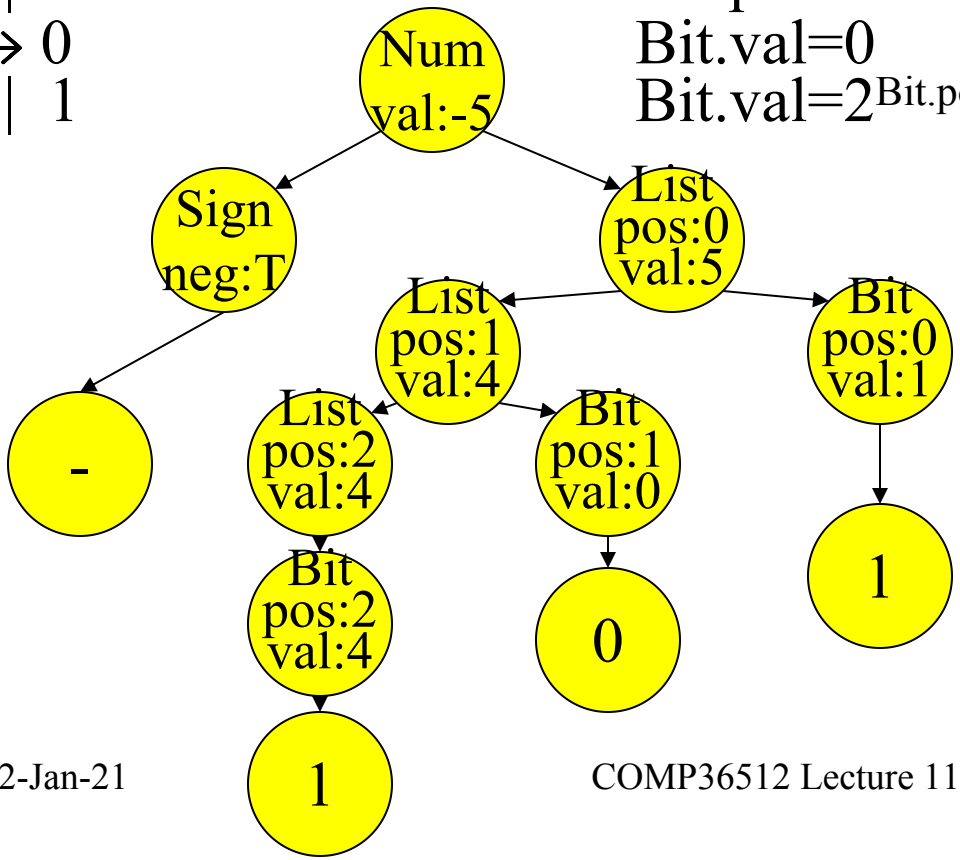


What kind of attribute is "type"? What about "in"?

In practice, we don't need to do all this with variable declarations! We can use a symbol table (for more see lecture 11). There, all we need to do is to add a new entry into a symbol table. E.g.: *addtype(id.type, L.in)*

Another Example (signed binary numbers)

Number→Sign List	List.pos=0; If Sign.neg then Number.val=-List.val else Number.val=List.val
Sign→ +	Sign.neg=false
| −	Sign.neg=true
List→List$_1$ Bit	List$_1$.pos=List.pos+1; Bit.pos=List.pos; List.val= List$_1$.val+Bit.val
| Bit	Bit.pos=List.pos; List.val=Bit.val
Bit → 0	Bit.val=0
| 1	Bit.val=$2^{\text{Bit.pos}}$



- Number and Sign have one attribute each. List and Bit have two attributes each.
- Val and neg are synthesised attributes; pos is an inherited attribute.
- What about an evaluation order? Knuth suggested a data-flow model of evaluation with independent attributes first. Implies a dependency graph!

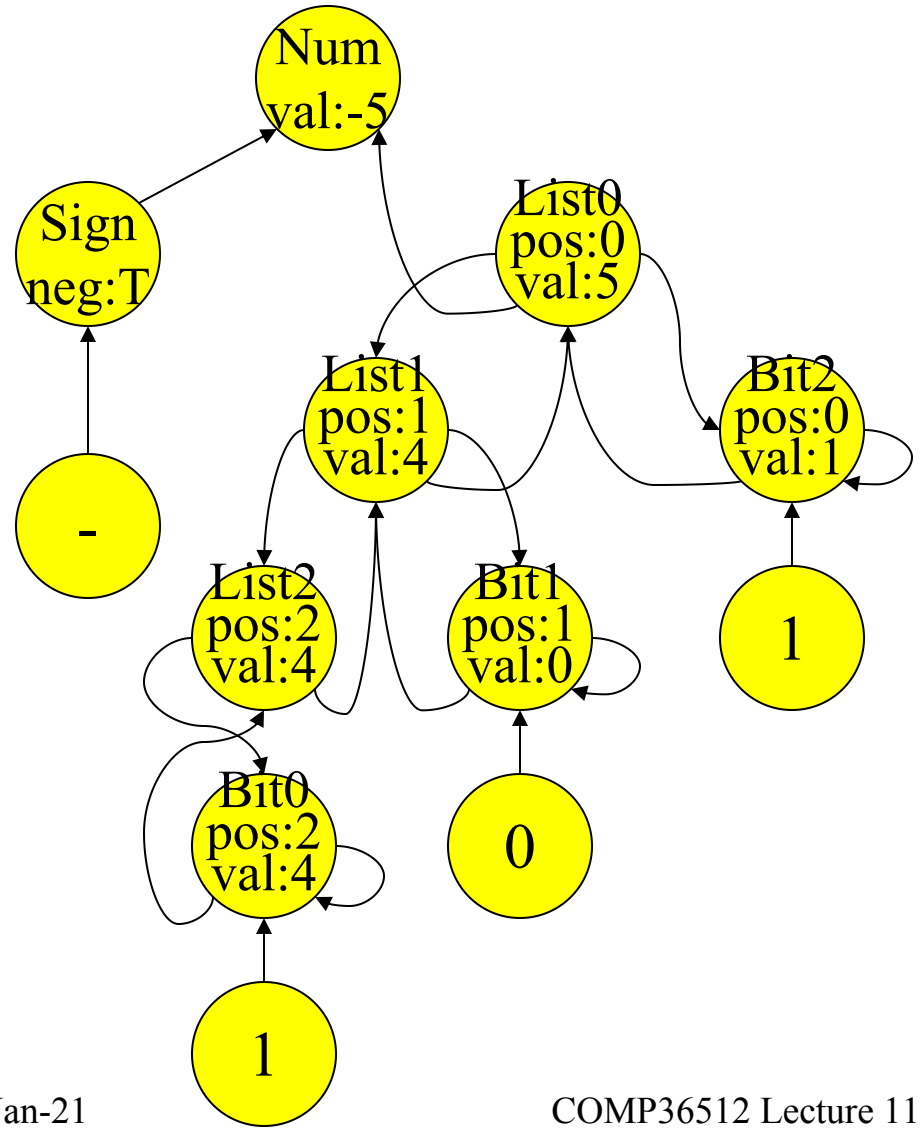
Attribute dependence graph

If an attribute at one node depends on an attribute of another node then the former must be evaluated before the latter.

Attribute Dependence Graph:

- Nodes represent attributes; edges represent the flow of values.
- Graph is specific to the parse tree (can be built alongside and its size is related to the size of the parse tree).
- Evaluation order:
  - Parse tree methods: use a topological sort (any ordering of the nodes such that edges go only from the earlier nodes to the later nodes: sort the graph, find independent values, then walk along graph edges): cyclic graph fails!
  - Rule-based methods: the order is statically predetermined.
  - Oblivious methods: use a convenient approach independent of semantic rules.
- Problem: how to deal with cycles.
- Another issue: complex dependencies.

Binary Numbers example: dependency graph



A topological order:

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

Using attribute grammars in practice

- Generic attribute grammars have seen limited practical use:
  - Complex local context handling is easy.
  - Non-local context-sensitive issues need a lot of supporting rules. Naturally, one tends to think about global tables… all this takes time…
- Still, there is some research and they have seen applications in structured editors, representation of structured documents, etc... (e.g, see attribute grammars and XML)
- In practice, a simplified idea is used:

  ***ad hoc* syntax directed translation**:

  - **S-attribute grammars**: only synthesized attributes are used. Values flow in only one direction (from leaves to root). It provides an evaluation method that works well with bottom-up parsers (such as those described in previous lectures).
  - **L-attribute grammars**: both synthesized and inherited attributes can be used, but there are certain rules that limit the use of inherited attributes.

Example (S-attribute grammar) (lect. 9, slide 5)

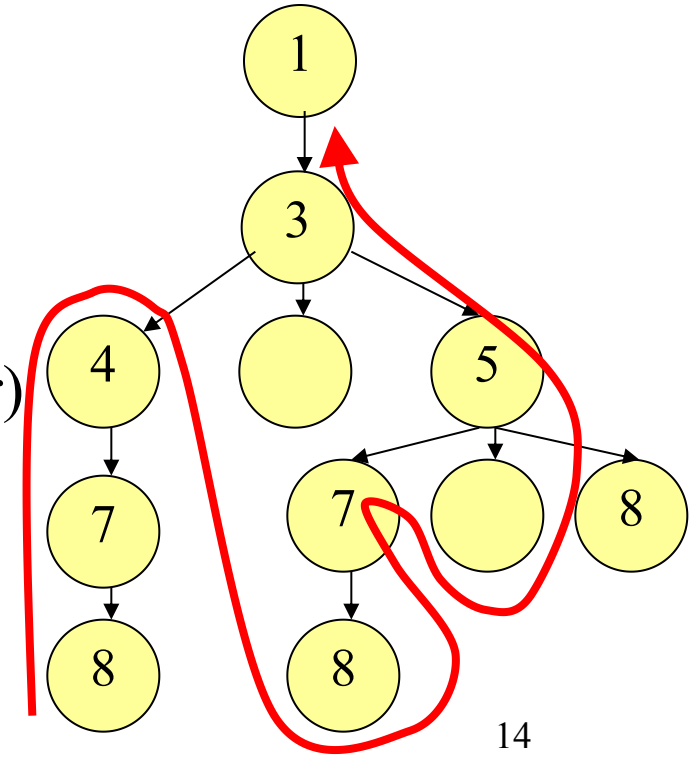
1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Using LR parsing (see table in lecture 9, slide 5), the reductions for the string 19-8*2 make use of the rules (in this order): 8, 7, 4, 8, 7, 8, 5, 3, 1. The attributes of the left-hand side are easily computed in this order too!

Assume we specify for each symbol the attribute *val* to compute the value of an expression (semantic rules are omitted, but are obvious).

Parse tree:

The red arrow shows the flow of values, which is the order that rules (cf. number) are discovered!



Finally…

- Another example: (signed binary numbers again)

Number→Sign List	Number = Sign.neg*List.val
Sign→ +	Sign.neg=1
| –	Sign.neg=–1
List→List$_1$ Bit	List.val=2*List1.val+Bit.val
| Bit	List.val=Bit.val
Bit → 0	Bit.val=0
| 1	Bit.val=1

- An L-attribute grammar: (see Aho2, example 5.8)

T → F Q	Q.inh = F.val
Q → * F Q$_1$	Q$_1$.inh = Q.inh * F.val

- Much of the (detailed) information in the parse tree is not needed for subsequent phases, so a condensed form (where non-terminal symbols are removed) is used: the **abstract syntax tree**.
- **Reading:** Aho2, Chapter 5; Aho1, pp.279-287; Grune pp.194-210; Hunter pp.147-152 (too general), Cooper Sec.4.1-4.4 (good discussion).