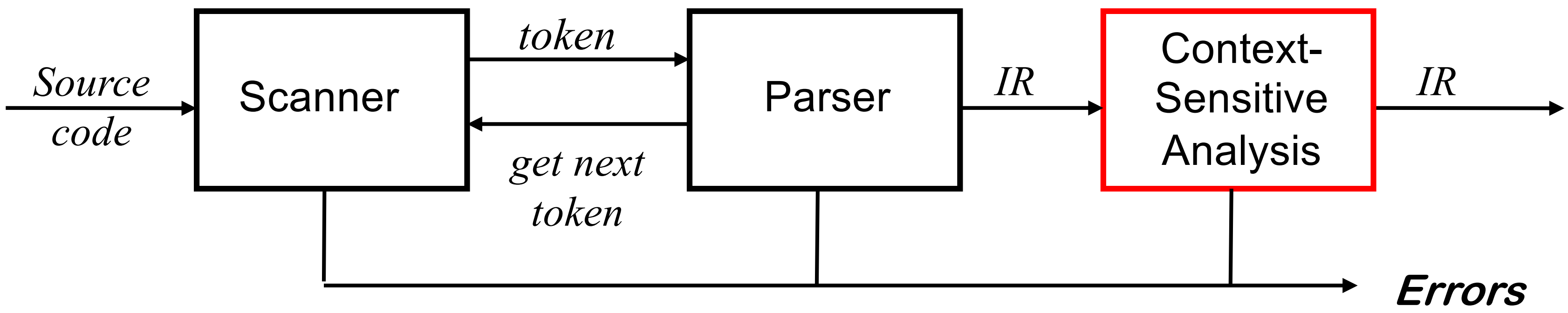
CMPSC 160
Translation of Programming Languages

Lecture 8: Context-Sensitive (Semantics) Analysis

Context-sensitive (semantics) analysis



- One of the jobs of the compiler front-end is to reject ill-formed inputs.
- This is usually done in three stages.
  - Lexical analysis: detects inputs with illegal lexical syntax.
  - Parsing: detects inputs with ill-formed syntax (no parse-tree).
  - Semantic analysis: catch 'all' remaining errors.
- Why do we need a separate semantic analysis phase at all?
  - Some language constraints are not expressible using CFGs (too complicated). The situation is similar to the split between lexing and parsing: not everything about syntactic well-formedness can be expressed by regular expressions & FSAs, so we use CFGs later.

What kind of errors can not be found with parsing?

- Can you think of any?

What kind of errors can not be found with parsing?

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

fee()
{
   int f[3], g[4], h, i,  j, k;
   char *p;
   call 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?

- declared g[4], 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"

What kinds of checks does semantic analysis do?

Some examples. The precise requirements depend on the language.

- All identifiers declared before use?
- Are all types correctly declared?
- Do the inheritance relationships make sense?
- Are classes and variables defined only once?
- Methods defined only once?
- Are private methods and members only used within the defining class?
- Stupid operations like cosine(true) or ”hello”/7?.

Why Lexing/Parsing is not good enough?

- The compiler must build up a large base of knowledge about the detailed computation encoded in the input program.
  - parsing level: only variable name; we just need to know it is a variable. Nothing beyond that.

- It must know what **values** are represented, where they reside, and how they flow from name to name.

- All of these facts can be derived from the source code. The compiler must perform deeper analysis than is typical for a scanner or a parser, the context-sensitive analysis.

- These kinds of analysis are either performed alongside parsing or in a post pass that traverses the IR produced by the parser.

Caveat

- When we say that semantic analysis catches ’all’ remaining errors, that does not include application-specific errors. It means catching errors that violate the well-formedness constraints that the language itself imposes.
- Naturally, those constraints are chosen by the language designers with a view towards efficient checkability by the compiler.

Rice’s theorem and undecidability

- **Rice’s theorem.** No interesting property of programs (more precisely: program execution) is decidable.

- That means for essentially any property that programs might have (e.g. does not crash, terminates, loops forever, uses more than 1782349 Bytes of memory) there cannot be a perfect checker, i.e., a program that determines with **perfect accuracy** whether the chosen property holds of any input program or not.

- Informally, one may summaries Rice’s theorem as follows: **to work out with 100% certainty what programs do, you have to run them (with the possibility of non-termination), there is no shortcut.**

Rice's theorem and undecidability

- Not all hope is lost!
- We can approximate a property of interest, and our approximation will have either false positives or false negatives (or both).

Rice's theorem and undecidability

- So, our semantic analysis must approximate.
- A compiler does this in a conservative way ("erring on the side of caution"): every program the semantic analysis accepts is guaranteed to have to properties that the semantic analysis check for, but the semantic analysis **will reject a lot of safe programs (having the required property).**
- Example: Our semantic analysis guarantees that programs never try to multiply an integer and a string like cosine("hello").
- Is the following program is safe?
  - if (x*x = -1) { y = 3 / "hello" } else y = 3 / 43110 }
- Yet any typing system in practical use will reject it. (Why?)

Semantic Analysis V.S. Lexing/Parsing

For lexing and parsing we proceeded in two steps.

1. Specify our expectation formally (RE for lexing, CFGs for parsing)
2. Invented algorithm to check our program given in (1): DFA to decide REs, (top-down/bottom-up) parser to decide CFGs.

For semantic analysis such a nice separation between specification and algorithm is difficult / an open problem.

- It seems hard to express our expectation (or constraints) independent from giving an algorithm that checks for them.
- The whole session on semantic analysis will be more **superficial** than those on lexing/parsing.

Different Semantic Analysis

- Semantics analysis is not one analysis, but a set of analysis.
- The compiler will have an abstraction for each of these categories of analysis.
- It uses abstractions that represent some aspect of the code, such as a type system, a storage map, or a control-flow graph.
  - For example, with **type system**, to variables of type string, we can apply operations such as *println*, but we cannot multiply two strings/
  - It must understand the program's **name space**: the kinds of data represented in the program, the kinds of data that can be associated with each name and each expression, and the mapping from a name's appearance in the code back to a specific instance of that name.
  - It must understand the **flow of control**, both within procedures and across procedures.

Commonality of Different Semantics Analysis

What kind of error detections are semantics analysis?

- Analysis depend on **values**, not just tokens
  - Analysis depend on attributes of tokens
- Analysis involve **non-local** information
  - variable declarations, procedures
- Analysis may involve **computation**

How can we answer these questions?

- Use formal methods
  - Context-sensitive grammars
  - Attribute grammars (semantic rules do not have side effects)
- Use ad-hoc techniques
  - Symbol tables
  - Syntax-directed translation (use semantic rules that can have side effects)

Attribute Grammars

- **Attribute grammar:** An attribute grammar consists of a context-free grammar augmented by **a set of rules** that specify **computations**.
  - We must decide what **attributes** each node (T/NT) needs.
  - We must elaborate the productions with **rules** that define values for these attributes in terms of the values of other attributes.
  - The rules are **functional**; they imply **no specific evaluation order** and they define each attribute's value uniquely.

Build Attribute Grammars from CFG: An Example

- A context-free grammar for signed binary numbers (SBN)

Number	$\rightarrow$	*Sign List*
Sign	$\rightarrow$	$+$
	$\mid$	$-$
List	$\rightarrow$	*List Bit*
	$\mid$	*Bit*
Bit	$\rightarrow$	0
	$\mid$	1

- *SBN* generates all signed binary numbers, such as -101, +11, -01, and +11111001100.
- It excludes unsigned binary numbers, such as 10.

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

Define Attributes for Grammar Symbols

- The compiler writer determines **a set of attributes** for each symbol in the grammar.

$$
\begin{array}{lcl}
\textit{Number} & \rightarrow & \textit{Sign List} \\
\textit{Sign} & \rightarrow & + \\
 & | & - \\
\textit{List} & \rightarrow & \textit{List Bit} \\
 & | & \textit{Bit} \\
\textit{Bit} & \rightarrow & 0 \\
 & | & 1
\end{array}
$$

Symbol	Attributes
Number	`value`
Sign	`negative`
List	`position, value`
Bit	`position, value`

- Simpler attribute grammars can solve this particular problem; we have chosen this one to demonstrate particular features of attribute grammars.

Define Rules to Facilitate the Information Flow among These Attributes

- The compiler writer also designs a set of rules to compute their values
  - These rules are functional.
  - Each rule implicitly defines a set of dependences.

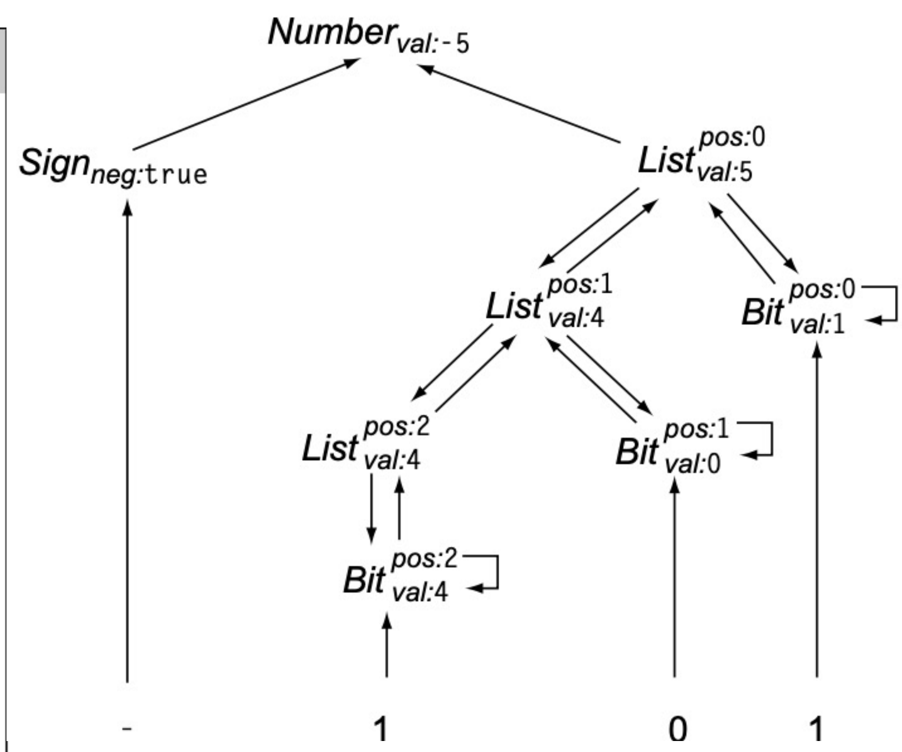
	Production	Attribution Rules
1	$Number \rightarrow Sign\ List$	$List.\mathtt{position} \leftarrow 0$ $\mathtt{if}\ Sign.\mathtt{negative}$ $\quad \mathtt{then}\ Number.\mathtt{value} \leftarrow -List.\mathtt{value}$ $\quad \mathtt{else}\ Number.\mathtt{value} \leftarrow List.\mathtt{value}$
2	$Sign \rightarrow +$	$Sign.\mathtt{negative} \leftarrow \mathtt{false}$
3	$Sign \rightarrow -$	$Sign.\mathtt{negative} \leftarrow \mathtt{true}$
4	$List \rightarrow Bit$	$Bit.\mathtt{position} \leftarrow List.\mathtt{position}$ $List.\mathtt{value} \leftarrow Bit.\mathtt{value}$
5	$List_0 \rightarrow List_1\ Bit$	$List_1.\mathtt{position} \leftarrow List_0.\mathtt{position} + 1$ $Bit.\mathtt{position} \leftarrow List_0.\mathtt{position}$ $List_0.\mathtt{value} \leftarrow List_1.\mathtt{value} + Bit.\mathtt{value}$
6	$Bit \rightarrow 0$	$Bit.\mathtt{value} \leftarrow 0$
7	$Bit \rightarrow 1$	$Bit.\mathtt{value} \leftarrow 2^{Bit.\mathtt{position}}$

Subscripts are added to grammar symbols whenever a specific symbol appears multiple times in a single production.

Attribute Dependence Graph

- Example: Attribute Dependence Graph for -101.
- Edges in the graph follow the flow of values in the evaluation of a rule.

	Production	Attribution Rules
1	*Number* → *Sign List*	*List*.position ← 0 if *Sign*.negative then *Number*.value ← - *List*.value else *Number*.value ← *List*.value
2	*Sign* → +	*Sign*.negative ← false
3	*Sign* → -	*Sign*.negative ← true
4	*List* → *Bit*	*Bit*.position ← *List*.position *List*.value ← *Bit*.value
5	$List_0$ → $List_1$ *Bit*	$List_1$.position ← $List_0$.position+1 *Bit*.position ← $List_0$.position $List_0$.value ← $List_1$.value + *Bit*.value
6	*Bit* → 0	*Bit*.value ← 0
7	*Bit* → 1	*Bit*.value ← $2^{Bit.position}$



Based on the dependence graph, we can define an evaluation order to calculate all attributes.

Categories of Attributes

- Attribute types → information flow directions → evaluation order

- We distinguish between attributes based on the direction of value flow.
  - *Synthesized attributes* are defined by bottom-up information flow
    - A synthesized attribute can draw values from the node itself, its descendants in the parse tree, and constants.
  - *Inherited attributes* are defined by top-down and lateral information flow
    - an inherited attribute can draw values from the node itself, its parent and its siblings in the parse tree, and constants.

Categories of Attributes

	Production	Attribution Rules

1	*Number* → *Sign List*	*List* . `position` ← `0`
		`if` *Sign* . `negative`
		`then` *Number* . `value` ← - *List* . `value`
		`else` *Number* . `value` ← *List* . `value`

2	*Sign* → +	*Sign* . `negative` ← `false`
3	*Sign* → -	*Sign* . `negative` ← `true`
4	*List* → *Bit*	*Bit* . `position` ← *List* . `position` *List* . `value` ← *Bit* . `value`
5	$List_0 \rightarrow List_1\ Bit$	$List_1$. `position` ← $List_0$. `position` + `1` *Bit* . `position` ← $List_0$. `position` $List_0$. `value` ← $List_1$. `value` + *Bit* . `value`
6	*Bit* → 0	*Bit* . `value` ← `0`
7	*Bit* → 1	*Bit* . `value` ← $2^{Bit.position}$

What type of attributes are them?

- Value
- Negative
- position

S-Attributed Grammars

- A grammar that uses only synthesized attributes is called an:

  S-attributed grammar

  - S-attributed grammars can be evaluated in a single bottom-up pass of the parse tree.

- LR parsers can easily deal with S-attributed grammars without explicitly building the parse tree (huge memory consumption).
  - Store the attributes of the symbols in the parser stack
  - When a reduce action is taken
    - Symbols in the RHS of the production and their attributes are already in the stack
    - Compute the synthesized attributes of the symbol in the LHS of the production using the attributes of the symbols on the RHS

L-Attributed Grammars

- *L-attributed grammar :* If inherited attribute of a symbol is computed using the inherited attributes of its parent and attributes of symbols on its left in the production, then the grammar is called an:
- Every S-attributed grammar is also L-attributed.
- L-attributed grammars can be evaluated using a **depth-first traversal** of the parse tree and can be incorporated conveniently in LL parsers.

The nodes in this algorithm are the nodes of the parse tree

Start the depth-first traversal by calling the **dfsvisit** on the root of the parse tree

```
procedure dfsvisit(n: node)
begin
   for each child m of n from left to right do
      evaluate inherited attributes of m;
      dfsvisit(m);
   endfor
   evaluate synthesized attributes of n
end
```

Evaluator Generator and Evaluator Methods

For more general cases, the compiler writer must create an evaluator as an implementation.

- an ad hoc program
- or by using an **evaluator generator**—the more attractive option.
- This indeed is the attraction of attribute grammars.

Dependence-based methods

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

Oblivious methods

- Evaluate nodes in some pre-selected order repeatedly
- e.g., repeated right-to-left passes, and alternating left-to-right and right-to-left passes.

More details can be found in book 4.3.1.