CS 415: Lecture 12

- Context sensitive analysis - syntax-directed translation

Revised Schedule

- HW2: Wed 3/8
- Midterm: Wed 3/22
- Project phase 3: Mon 4/3 (tentative)
- Project phase 4: Wed 4/26 (tentative)
Where We Are

front end

lexical analyzer  →  syntax analyzer  →  semantic analyzer

symbol table

Intermediate code generator  →  code optimizer  →  code generator

back end

error handler

Context-Sensitive Analysis

- **Semantic analysis is typically context-sensitive**
- **Example questions**
  - Is $x$ a scalar, array, or function?
  - Is $x$ declared before it is used?
  - Are any names declared by not used?
  - Which declaration of $x$ does a particular use reference?
  - Is an expression type-consistent?
  - Does the dimension of an array reference match the declaration?
  - Is $p$ a pointer in a dereference $*p$?
  - Is an array reference in bounds?
  - Does function $foo$ produce a constant value?
- **These cannot be answered with a context-free grammar!**
Context-Sensitive Analysis

- Context-sensitive analysis is hard because
  - Need non-local information
  - Answers depend on values, not on syntax
  - Answers may involve computation
- One possible approach: write context-sensitive grammars and parse
  - General problem is P-space complete
  - Haven't found useful subclass
- Our approach
  - Syntax-directed definition (attribute grammars)
  - Translation schemes (attr. grammars with attr. evaluation order)

Attribute Grammars

- Formal framework based on grammar and parse tree used to specify and compute context-sensitive properties
- Idea: attribute the parse tree
  - Can add attributes (fields) to each node
  - Specify equations (semantic actions) to define values of attributes
  - Success of the parse can be based, in part, on attribute values
- Attribute grammars are very general.
  - Some example uses include: infix to postfix translation, type checking, construct intermediate representation, desk calculator, code generation
**Example**

nonterminals: N, B  
attributes: B.val, N.val  

productions semantic actions

\[
\begin{align*}
B & \rightarrow 1 & B \text{.val} = 1 \\
B & \rightarrow 0 & B \text{.val} = 0 \\
B_1 & \rightarrow B_2 1 & B_1 \text{.val} = 2*B_2 \text{.val} + 1 \\
B_1 & \rightarrow B_2 0 & B_1 \text{.val} = 2*B_2 \text{.val} \\
N & \rightarrow B & N \text{.val} = B \text{.val}
\end{align*}
\]

Attributes may depend on attributes of parent, siblings, or children.
Example

```
Example

Rutgers University, DCS

Attribute Grammars

- Generalization of context-free grammars
- Each grammar symbol has an associated set of attributes
- Augment grammar with rules defining attribute values - semantic actions. Semantic actions may have side effects.
- Two types of attributes
  - Inherited attributes: Values computed from attributes of parent (lhs non-terminal) or siblings (non-terminals on rhs of same production)
  - Synthesized attributes: Values computed from attributes of children

Rutgers University, DCS
```
Attribute Grammars

- Values can be associated with terminals (tokens) returned by the scanner
- Distinguished non-terminal cannot have inherited attributes
- Evaluation context: always within focus of a single production
  - Dependence edges go only one level up or down or across in parse tree

Example

Identifiers with no letters repeated
(e.g., moon - illegal, money - legal)

\[
D \rightarrow I \quad \text{I.str} = \{\}; \text{D.val} = \text{I.val};
\]
accept, if D.val ! = error

\[
I_1 \rightarrow L \ I_2 \quad \text{L.str} = I_1\text{.str}; \text{I_2.str} = \text{L.val};
\]
I_1.val = I_2.val;

\[
I \rightarrow L \quad \text{L.str} = \text{I.str}; \text{I.val} = \text{L.val};
\]

\[
L \rightarrow a \mid b \mid \ldots \mid z
\]
L.val = concatenation of val returned by scanner with L.str, if this character is not a repeated letter, else error.

(note: any comparison to error returns error.)
Parse Tree of abc

Decorated Parse Tree
Example - aba

Compiler Example

symb is the symbol table gathered from the declarations and checked in the statements.
S-attributed Grammars

- S-attributed grammars: all attributes are synthesized
- Easy to interleave with shift-reduce parsing by using a parallel stack for attribute values
  - Evaluate attributes when do a reduction
- Important: can code semantic functions *a priori*, because know all the handles from the grammar, so know where the associated attributes will be in the semantic value stack when a reduction is about to take place.

Semantic Value Stack

```
stacks   input
$        1 + 2 $
$        $ 
$ int-const $ + 2 $
$ 1      $ 
$ E      + 2 $
$ 1      $ 
$ E+     2 $
$ E + E  $ 
$ 1 ε 2  $ 
$ E + E  $ 
$ E      $ 
$ 3      $ 
```

Rutgers University, DCS 17 CS 415: Compilers
Inherited Attributes

- May be more difficult to compute inherited attributes - need to figure out dependency and compute accordingly
- Semantic actions define partial dependency graph
- Attribute dependency graph
  - Nodes represent attributes
  - Edges represent the flow of values
  - Graph is specific to parse tree
  - Size is related to parse tree's size
  - Can be built alongside parse tree
- Dependency graph must be acyclic

Attribute Dependency Graph

- Evaluation order
  - Topological sort the dependency graph to order semantic actions
  - Evaluate using this order
- Order depends on both the grammar and the particular input string
**Example Dependency Graph**

**A More Involved Example**

- See supplementary slides
Evaluation Methods (ASU’s Taxonomy)

- **Parse-tree methods**
  - Build parse tree
  - Build dependency graph
  - Topological sort
  - Evaluate
- **Rule-based methods**
  - Analyze rules at compiler-generation time
  - Determine a static ordering at that time
  - Evaluate nodes in that order at compile time
- **Ad-hoc methods**
  - Ignore the parse tree and grammar
  - Choose a “convenient” order and use it

Abstract Syntax Trees

- **May want to perform (some) semantic analysis, optimization, and translation separate from parsing**
- **Use Abstract Syntax Tree (AST) as intermediate language**
  - Shows the abstract syntax of the language (only keeps non-terminals which have some meaning attached)
  - Different from concrete syntax with punctuation, trivial productions (e.g., \( E \to T \to F \to id \))
  - Compilers can manipulate the abstract syntax once concrete syntax has been checked
Example

\[ 2 + 3 \]

![Parse tree](image)

Abstract syntax tree

Parse tree

AST Construction

- Can use syntax directed translation to build AST
- For example

<table>
<thead>
<tr>
<th>Productions</th>
<th>Semantic Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E' ::= E )</td>
<td>( E'.ptr \leftarrow E.ptr )</td>
</tr>
<tr>
<td>( E ::= E_1 + T )</td>
<td>( E ptr \leftarrow createNode(&quot;+&quot;, E_1 ptr, T ptr) )</td>
</tr>
<tr>
<td>( E ::= T )</td>
<td>( E ptr \leftarrow T ptr )</td>
</tr>
<tr>
<td>( T ::= id )</td>
<td>( T ptr \leftarrow createLeaf(&quot;id&quot;, id symEntry) )</td>
</tr>
</tbody>
</table>
L-Attributed Definitions

- AST construction is performed using S-attributed definitions
- A common class of syntax-directed definitions, called L-attributed definitions, has both synthesized attributes and inherited attributes but can be supported by both top-down (LL(1)) and (in most cases) bottom-up (LR(1)) parsers
- A syntax-directed definition is L-attributed if each inherited attribute of $X_j$, $1 \leq j \leq n$, on the right side of $A \rightarrow X_1 X_2 \ldots X_n$, depends only on
  - The attributes of the symbols $X_1 X_2 \ldots X_{j-1}$ to the left of $X_j$ in the production and
  - The inherited attributes of $A$
- Book discusses how to implement these definitions. We're not going to discuss them further