Lesson 10

- Continuation of the course
- Syntax-Directed Translation (1)
Continuation of the course

- [Nov-Dec 2014] 22 h
  - Introduction to compilers
  - Lexical analysis
  - Parsing

- [Feb-May 2015] ~50 h
  - Syntax directed translation
  - Intermediate code generation
  - Code generation

- Concepts of Programming Languages
- <to be detailed...>

- May 27-29: 2nd Mid-Term Exam
- Can be taken by everybody
Continuation of the course (2)

- Office hours: **Wednesday, 4-6 pm**
- 9 Credits vs. 12 Credits: still a problem for somebody?
- Important: no lectures on
  - Friday, March 6
  - Tuesday, March 17
  - Friday, March 20
- Need to recover several lectures with 3 lectures per week
  - Possible days and hours: **Thursday, 2-4 pm**
The Structure of the Front-End

Source Program (Character stream) → Lexical analyzer → Token stream → Parser (Syntax-directed translator) → Intermediate representation

Develop parser and code generator for translator

Syntax definition (BNF grammar) → IR specification
Syntax-Directed Translation

• Briefly introduced in the first lectures
• General technique to “manipulate” programs, based on context-free grammars
• Tightly bound with parsing
• Will be used for static analysis (type checking) and (intermediate) code generation
• Several other uses:
  – Generation of abstract syntax trees
  – Evaluation of expressions
  – Implementation of Domain Specific Languages (see example on typesetting math formulas in the book)
  – ...
• Partly supported by parser generators like Yacc
Syntax-Directed Definitions

• A *syntax-directed definition* (or *attribute grammar*) binds a set of *semantic rules* to productions
• Terminals and nonterminals have *attributes* holding values, which are set by the semantic rules
• A *depth-first (postorder) traversal* algorithm traverses the parse tree executing semantic rules to assign attribute values
• After the traversal is complete the attributes contain the translated form of the input
Example: evaluating expressions with *synthesized* attributes

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow E \ n$</td>
<td>$\text{print}(E.\text{val})$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.\text{val} := E_1.\text{val} + T.\text{val}$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.\text{val} := T.\text{val}$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F$</td>
<td>$T.\text{val} := T_1.\text{val} * F.\text{val}$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T.\text{val} := F.\text{val}$</td>
</tr>
<tr>
<td>$F \rightarrow ( E )$</td>
<td>$F.\text{val} := E.\text{val}$</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>$F.\text{val} := \text{digit}$.\text{lexval}$</td>
</tr>
</tbody>
</table>

*A Syntax-Directed Definition (SDD) or Attribute Grammar*
Example: An Annotated Parse Tree

```
E.val = 16
  |
  v
E.val = 14
  |
  v
E.val = 9  T.val = 5
  |
  v
T.val = 9  F.val = 5
  |
  v
F.val = 9

9 + 5 + 2 n
```

Productions

- $L \rightarrow E \text{ n}$
- $E \rightarrow E_1 + T$
- $E \rightarrow T$
- $T \rightarrow T_1 * F$
- $T \rightarrow F$
- $F \rightarrow ( E )$
- $F \rightarrow \text{ digit}$
Annotating a Parse Tree with Depth-First Traversals

**procedure** visit\( (n : node) \);  
**begin**  
   **for** each child \( m \) of \( n \), from left to right **do**  
     visit\( (m) \);  
     evaluate semantic rules at node \( n \)  
**end**
Depth-First Traversals (Example)

Semantic Rules

\[
\begin{align*}
\text{print}(E.\text{val}) \\
E.\text{val} &:= E_1.\text{val} + T.\text{val} \\
E.\text{val} &:= T.\text{val} \\
T.\text{val} &:= T_1.\text{val} \times F.\text{val} \\
T.\text{val} &:= F.\text{val} \\
F.\text{val} &:= E.\text{val} \\
F.\text{val} &:= \text{digit}.\text{lexval}
\end{align*}
\]
Attributes

• Each grammar symbol can have any number of attributes

• Attribute values typically represent
  – Numbers (literal constants)
  – Strings (literal constants)
  – Memory locations, such as a frame index of a local variable or function argument
  – A data type for type checking of expressions
  – Scoping information for local declarations
  – Intermediate program representations
Synthesized vs. Inherited Attributes

• Given a production
  
  \[ A \rightarrow \alpha \]
  
  then each semantic rule is of the form
  
  \[ b := f(c_1, c_2, \ldots, c_k) \]
  
  where \( f \) is a function and \( c_i \) are attributes of \( A \) and \( \alpha \), and either
  
  – \( b \) is a \textit{synthesized} attribute of \( A \)
  
  – \( b \) is an \textit{inherited} attribute of one of the grammar symbols in \( \alpha \)
Synthesized Versus Inherited Attributes (cont’d)

Production                Semantic Rule

\[ D \rightarrow T \; L \]  \[ L.\text{in} = T.\text{type} \]
\[ T \rightarrow \text{int} \]  \[ T.\text{type} := \text{‘integer’} \]
\[ \ldots \]  \[ \ldots \]
\[ L \rightarrow \text{id} \]  \[ \ldots := L.\text{in} \]
S-Attributed Definitions

• A syntax-directed definition that uses synthesized attributes exclusively is called an *S-attributed definition (or S-attributed grammar)*

• A parse tree of an S-attributed definition can be annotated with a single bottom-up traversal

• [Yacc/Bison only support S-attributed definitions]
Example: generation of Abstract Syntax Trees

• A parse tree is called a *concrete syntax tree*

• An *abstract syntax tree (AST)* is defined by the compiler writer as a more convenient intermediate representation
### S-Attributed Definitions for Generating Abstract Syntax Trees

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.nptr := \text{mknode}(\text{‘+’}, E_1.nptr, T.nptr)$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 - T$</td>
<td>$E.nptr := \text{mknode}(\text{‘-’}, E_1.nptr, T.nptr)$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.nptr := T.nptr$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * \text{id}$</td>
<td>$T.nptr := \text{mknode}(\text{‘*’}, T_1.nptr, \text{mkleaf(id, id.entry)})$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 / \text{id}$</td>
<td>$T.nptr := \text{mknode}(\text{‘/’}, T_1.nptr, \text{mkleaf(id, id.entry)})$</td>
</tr>
<tr>
<td>$T \rightarrow \text{id}$</td>
<td>$T.nptr := \text{mkleaf(id, id.entry)}$</td>
</tr>
</tbody>
</table>
Generating Abstract Syntax Trees

Synthesize AST

```
E.nptr
    +
T.nptr
    *
  id
id

id
+ id

id
* id
```
Example Attribute Grammar with Synthesized + Inherited Attributes

- Grammar generating declaration of typed variables
- The attributes add typing information to the symbol table via side effects

### Production

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow TL$</td>
<td>$L\text{.in} := T\text{.type}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{int}$</td>
<td>$T\text{.type} := 'integer'$</td>
</tr>
<tr>
<td>$T \rightarrow \text{real}$</td>
<td>$T\text{.type} := 'real'$</td>
</tr>
<tr>
<td>$L \rightarrow L_1, \text{id}$</td>
<td>$L_1\text{.in} := L\text{.in}; \text{addtype}(\text{id}.\text{entry}, L\text{.in})$</td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\text{addtype}(\text{id}.\text{entry}, L\text{.in})$</td>
</tr>
</tbody>
</table>

Synthesized: $T\text{.type}, \text{id}.\text{entry}$
Inherited: $L\text{.in}$
Evaluation order of attributes

- In presence of inherited attributes, it is not obvious in which order
  the attributes can be evaluated

Grammar generating declaration of typed variables
- The attributes add typing information to the symbol table
  via side effects

Production | Semantic Rule
\[\text{Production} \quad \text{Semantic Rule} \]
\[D \rightarrow TL \quad L\text{.in} := T\text{.type} \]
\[T \rightarrow \text{int} \quad T\text{.type} := \text{‘integer’} \]
\[T \rightarrow \text{real} \quad T\text{.type} := \text{‘real’} \]
\[L \rightarrow L_1 , \text{id} \quad L_1\text{.in} := L\text{.in}; \text{addtype}(\text{id}\text{.entry}, L\text{.in}) \]
\[L \rightarrow \text{id} \quad \text{addtype}(\text{id}\text{.entry}, L\text{.in}) \]

Synthesized: \(T\text{.type, id}\text{.entry} \)
Inherited: \(L\text{.in} \)
Evaluation order of attributes

• In presence of inherited attributes, it is not obvious in what order the attributes should be evaluated

• Attributes of a nonterminal in a production may depend in an arbitrary way on attributes of other symbols

• The evaluation order must be consistent with such dependencies
Dependency Graphs for Attributed Parse Trees

\[ A \rightarrow X Y \]

\[ A.a := f(X.x, Y.y) \]

\[ X.x := f(A.a, Y.y) \]

\[ Y.y := f(A.a, X.x) \]
Dependency Graphs with Cycles?

- Edges in the dependency graph determine the evaluation order for attribute values
- Dependency graphs cannot be cyclic

\[
\begin{align*}
A.a & := f(X.x) \\
X.x & := f(Y.y) \\
Y.y & := f(A.a)
\end{align*}
\]

Error: cyclic dependence
Example Annotated Parse Tree

\[
D \rightarrow TL \quad L.in := T.type \\
T \rightarrow \text{int} \quad T.type := \text{‘integer’} \\
T \rightarrow \text{real} \quad T.type := \text{‘real’} \\
L \rightarrow L_1, id \quad L_1.in := L.in; \ addtype(id.entry, L.in) \\
L \rightarrow id \quad addtype(id.entry, L.in)
\]

\[
D \\
  \quad T.type = \text{‘real’} \quad L.in = \text{‘real’} \\
  \quad \quad \text{real} \quad L.in = \text{‘real’}, \quad id_3.entry \\
  \quad \quad \quad L.in = \text{‘real’}, \quad id_2.entry \\
  \quad \quad \quad \quad id_1.entry
\]
Example Annotated Parse Tree with Dependency Graph

\[ D \rightarrow TL \quad L_{.in} := T_{.type} \]
\[ T \rightarrow \text{int} \quad T_{.type} := \text{‘integer’} \]
\[ T \rightarrow \text{real} \quad T_{.type} := \text{‘real’} \]
\[ L \rightarrow L_1, \text{id} \quad L_{1.in} := L_{.in}; \text{addtype}(\text{id}.\text{entry}, L_{.in}) \]
\[ L \rightarrow \text{id} \quad \text{addtype}(\text{id}.\text{entry}, L_{.in}) \]
Evaluation Order

- A *topological sort* of a directed acyclic graph (DAG) is any ordering \( m_1, m_2, \ldots, m_n \) of the nodes of the graph, such that if \( m_i \rightarrow m_j \) is an edge, then \( m_i \) appears before \( m_j \).

- Any topological sort of a dependency graph gives a valid evaluation order of the semantic rules.
Example Parse Tree with Topologically Sorted Actions

Topological sort:
1. Get \text{id}_1.entry
2. Get \text{id}_2.entry
3. Get \text{id}_3.entry
4. \( T_1\.\text{type} = \text{'real'} \)
5. \( L_1\.\text{in} = \text{'real'} \)
6. \( \text{addtype}(	ext{id}_3\.\text{entry}, L_1\.\text{in}) \)
7. \( L_2\.\text{in} = L_1\.\text{in} \)
8. \( \text{addtype}(	ext{id}_2\.\text{entry}, L_2\.\text{in}) \)
9. \( L_3\.\text{in} = L_2\.\text{in} \)
10. \( \text{addtype}(	ext{id}_1\.\text{entry}, L_3\.\text{in}) \)
Evaluation Methods

- *Parse-tree methods* determine an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input.
- *Rule-base methods* the evaluation order is predetermined from the semantic rules.
- *Oblivious methods* the evaluation order is fixed and semantic rules must be (re)written to support the evaluation order (for example, S-attributed definitions).
L-Attributed Definitions

- A syntax-directed definition is *L-attributed* if each inherited attribute of $X_j$ on the right side of $A \rightarrow X_1 X_2 \ldots X_n$ depends only on
  1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$
  2. the inherited attributes of $A$

![Possible dependences of inherited attributes](image)

\[ A.a \quad X_1.x \quad X_2.x \]
L-Attributed Definitions (cont’d)

- L-attributed definitions allow for a natural order of evaluating attributes: depth-first and left to right

\[
A \rightarrow X Y
\]

\[
X.i := A.i \\
A \rightarrow A.s := Y.s \\
Y.i := X.s \\
\]

- Note: every S-attributed syntax-directed definition is also L-attributed (since it doesn’t have any inherited attribute)
Syntax-Directed Translation Schemes

- A *translation scheme* is a CF grammar embedded with *semantic actions*

\[
\text{rest} \rightarrow + \text{term} \{ \text{print("+")} \} \text{rest}
\]

Embedded semantic action
Syntax-Directed Translation Schemes

• Translation Schemes are an alternative notation for Syntax-Directed Definitions
• The semantic rules can be suitably embedded into productions
• SDT’s can always be implemented by building the parse tree first, and then performing the actions in left-to-right depth-first order
• In several cases they can be implemented during parsing, without building the whole parse tree first
Postfix Translation Schemes

- If the grammar is LR (thus can be parsed bottom-up) and the SDD is S-attributed (synthesized attributes only), semantic actions can be placed at the end of the productions.
- They are executed when the body is reduced to the head.
- These are called *postfix SDTs*.
Example Translation Scheme for Postfix Notation

\[
\begin{align*}
expr &\rightarrow expr + term & \{ \text{print(“+”) } \} \\
expr &\rightarrow expr - term & \{ \text{print(“-”) } \} \\
expr &\rightarrow term \\
term &\rightarrow 0 & \{ \text{print(“0”) } \} \\
term &\rightarrow 1 & \{ \text{print(“1”) } \} \\
\ldots & & \ldots \\
term &\rightarrow 9 & \{ \text{print(“9”) } \}
\end{align*}
\]
Example Translation Scheme (cont’d)

Translates 9-5+2 into postfix 95-2+
Implementation of Postfix SDTs

- Postfix SDTs can be implemented during LR parsing
- The actions are executed when reductions occur
- The attributes of grammar symbols can be put on the stack, together with the symbol or the state corresponding to it
- Since all attributes are synthesized, the attribute for the head can be computed when the reduction occurs, because all attributes of symbols in the body are already computed