Context-sensitive Analysis, II

*Ad-hoc syntax-directed translation, Symbol Tables, and Types*
Remember the Example from Last Lecture?

Grammar for a basic block

Let’s estimate cycle counts
• Each operation has a COST
• Add them, bottom up
• Assume a load per value
• Assume no reuse
Simple problem for an AG

Hey, this looks useful!
And Its Extensions

Tracking loads

• Introduced *Before* and *After* sets to record loads
• Added ≥ 2 copy rules per production
  → Serialized evaluation into execution order
• Made the whole attribute grammar large & cumbersome
The Moral of the Story

• Non-local computation needed lots of supporting rules
• Complex local computation was relatively easy

The Problems

• Copy rules increase complexity
  → Hard to understand and maintain
• Copy rules increase space requirements
  → Need copies of attributes
  → Can use pointers, but harder to understand
Addressing the Problem

If you gave this problem to a programmer at IBM

- Introduce a central repository for facts
- Table of names
  - Field in table for loaded/not loaded state
- Avoids all the copy rules, allocation & storage headaches
- All inter-assignment attribute flow is through table
  - Clean, efficient implementation
  - Good techniques for implementing the table (hashing, § B.3)
  - When its done, information is in the table!
  - Cures most of the problems
- Unfortunately, this design violates the functional paradigm
  - Do we care?
Remind ourselves of Compiler Phases

Different Phases of Project

Phase I: Scanner
Phase II: Parser
Phase III: Semantic Routines
Phase IV: Code Generator
The Realist’s Alternative

Ad-hoc syntax-directed translation

•  Associate a snippet of code with each production
•  At each reduction, the corresponding snippet runs
•  Allowing arbitrary code provides complete flexibility
  → Includes ability to do tasteless & bad things

To make this work

•  Need names for attributes of each symbol on lhs & rhs
  → Typically, one attribute passed through parser + arbitrary code
     (structures, globals, statics, …)
•  Need an evaluation scheme
  → Fits nicely into LR(1) parsing algorithm
Reworking the Example  *(with load tracking)*

<table>
<thead>
<tr>
<th>Block (_0) → Block (_1) Assign</th>
<th>Assign</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign → Ident = Expr ;</td>
<td>cost ← cost + COST(store);</td>
</tr>
<tr>
<td>Expr (_0) → Expr (_1) + Term</td>
<td>cost ← cost + COST(add);</td>
</tr>
<tr>
<td></td>
<td>Expr (_1) - Term</td>
</tr>
<tr>
<td></td>
<td>Term</td>
</tr>
<tr>
<td>Term (_0) → Term (_1) * Factor</td>
<td>cost ← cost + COST(mult);</td>
</tr>
<tr>
<td></td>
<td>Term (_1) / Factor</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
</tr>
<tr>
<td>Factor → ( Expr )</td>
<td>cost ← cost + COST(loadi);</td>
</tr>
<tr>
<td></td>
<td>Number</td>
</tr>
<tr>
<td></td>
<td>Identifier</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This looks simpler than the Attribute Grammar solution!

One missing detail: initializing cost
Reworking the Example (with load tracking)

\[
\begin{array}{|l|c|}
\hline
\text{Start} & \rightarrow \text{Init Block} \\
\text{Init} & \rightarrow \varepsilon \\
\text{Block}_0 & \rightarrow \text{Block}_1 \text{ Assign} \\
& | \text{Assign} \\
\text{Assign} & \rightarrow \text{Ident} = \text{Expr} \\
& | \text{cost} \leftarrow \text{cost} + \text{COST(store)}; \\
\hline
\end{array}
\]

... and so on as in the previous version of the example ...

- Before parser can reach Block, it must reduce Init
- Reduction by Init sets cost to zero

This is an example of splitting a production to create a reduction in the middle — for the sole purpose of hanging an action routine there!
Example — Building an Abstract Syntax Tree

- Assume constructors for each node
- Assume stack holds pointers to nodes

<table>
<thead>
<tr>
<th>Goal</th>
<th>Expr</th>
<th>Goal.node = E.node;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>Expr + Term</td>
<td>E&lt;sub&gt;0&lt;/sub&gt;.node = MakeAddNode(E&lt;sub&gt;1&lt;/sub&gt;.node, T.node);</td>
</tr>
<tr>
<td></td>
<td>Expr - Term</td>
<td>E&lt;sub&gt;0&lt;/sub&gt;.node = MakeSubNode(E&lt;sub&gt;1&lt;/sub&gt;.node, T.node);</td>
</tr>
<tr>
<td></td>
<td>Term</td>
<td>E.node = T.node;</td>
</tr>
<tr>
<td>Term</td>
<td>Term * Factor</td>
<td>T&lt;sub&gt;0&lt;/sub&gt;.node = MakeMulNode(T&lt;sub&gt;1&lt;/sub&gt;.node, F.node);</td>
</tr>
<tr>
<td></td>
<td>Term / Factor</td>
<td>T&lt;sub&gt;0&lt;/sub&gt;.node = MakeDivNode(T&lt;sub&gt;1&lt;/sub&gt;.node, F.node);</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
<td>T.node = F.node;</td>
</tr>
<tr>
<td>Factor</td>
<td>( Expr )</td>
<td>F.node = Expr.node;</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td>F.node = MakeNumNode(token);</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>F.node = MakeIdNode(token);</td>
</tr>
</tbody>
</table>
Reality

Most parsers are based on this *ad-hoc* style of context-sensitive analysis

Advantages

- Addresses shortcomings of Attribute Grammar paradigm
- Efficient, flexible

Disadvantages

- Must write the code with little assistance
- Programmer deals directly with the details

Most parser generators support a yacc/bison-like notation
Typical Uses

• Building a symbol table
  - Enter declaration information as processed
  - At end of declaration syntax, do some post processing
  - Use table to check errors as parsing progresses

• Simple error checking/type checking
  - Define before use → lookup on reference
  - Dimension, type, ... → check as encountered
  - Type conformability of expression → bottom-up walk
  - Procedure interfaces are harder
    ✦ Build a representation for parameter list & types
    ✦ Create list of sites to check
    ✦ Check offline, or handle the cases for arbitrary orderings
Symbol Tables

• For compile-time efficiency, compilers use symbol tables
  → Associates lexical names (symbols) with their attributes

• What items go in symbol tables?
  → Variable names
  → Defined constants
  → Procedure/function/method names
  → Literal constants and strings
  → Separate layout for structure layouts
    ♦ Field offsets and lengths

• A symbol table is a compile-time structure
• More after mid-term!
Attribute Information

- Attributes are internal representation of declarations
- Symbol table associates names with attributes
- Names may have different attributes depending on their meaning:
  - Variables: type, procedure level
  - Types: type descriptor, data size/alignment
  - Constants: type, value
  - Procedures: Signature (arguments/types), result type, etc.
Type Systems

- **Types**
  - Values that share a set of common properties
  - Defined by language (built-ins) and/or programmer (user-defined)

- **Type System**
  - Set of types in a programming language
  - Rules that use types to specify program behavior

- **Example type rules**
  - If operands of addition are of type integer, then result is of type integer
  - The result of the unary “&” operator is a pointer to the object referred to by the operand

- **Advantages**
  - Ensures run-time safety
  - Provides information for code generation
Type Checker

- Enforces rules of the type system
- May be strong/weak, static/dynamic

- Static type checking
  - Performed at compile time
  - Early detection, no run-time overhead
  - Not always possible (e.g., A[I], where I comes from input)

- Dynamic type checking
  - Performed at run time
  - More flexible, rapid prototyping
  - Overhead to check run-time type tags
Type expressions

• Used to represent the type of a language construct
• Describes both language and programmer types

• Examples
  → Basic types (built-ins) : integer, float, character
  → Constructed types : arrays, structs, functions
A simple type checker

Using a synthesized attribute grammar, we will describe a type checker for arrays, pointers, statements, and functions.

Grammar for source language:

\[
\begin{align*}
P &::= D ; E \\
D &::= D ; E \mid \text{id}: T \\
T &::= \text{char} \mid \text{integer} \mid \text{array}[\text{num}] \text{ of } T \mid \uparrow T \\
E &::= \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E[E] \mid E \uparrow
\end{align*}
\]

- Basic types \textit{char}, \textit{integer}, \textit{typeError}

- assume all arrays start at 1, \textit{e.g.},
  array [256] of char
  results in the type expression \textit{array}(1\ldots256,\text{char})

- \uparrow builds a pointer type, so \uparrow \text{integer}
  results in the type expression \textit{pointer}(\text{integer})
Partial attribute grammar for the type system

\[
\begin{align*}
D &::= \text{id: } T \\
T &::= \text{char} \\
T &::= \text{integer} \\
T &::= \text{\textasciitilde } T_1 \\
T &::= \text{array [num] of } T_1
\end{align*}
\]

\[
\begin{align*}
\{ &\text{addtype(id.entry, } T\text{.type) } \\
&\{ T\text{.type }\leftarrow \text{char} \} \\
&\{ T\text{.type }\leftarrow \text{integer} \} \\
&\{ T\text{.type }\leftarrow \text{pointer}(T_1\text{.type}) \} \\
&\{ T\text{.type }\leftarrow \text{array}(1\dotsc\text{num.val}, T_1\text{.type}) \}
\end{align*}
\]
Type checking expressions

Each expression is assigned a type using rules associated with the grammar.

\[
\begin{align*}
\text{E} &::= \text{literal} \quad \{ \text{E.type} \leftarrow \text{char} \} \\
\text{E} &::= \text{num} \quad \{ \text{E.type} \leftarrow \text{integer} \} \\
\text{E} &::= \text{id} \quad \{ \text{E.type} \leftarrow \text{lookup(id.entry)} \} \\
\text{E} &::= \text{E}_1 \mod \text{E}_2 \quad \begin{cases} &\text{E.type} \leftarrow \text{if E}_1.\text{type} = \text{integer} \text{ and} \\
&\text{E}_2.\text{type} = \text{integer} \text{ then integer} \\
&\text{else typeError} \end{cases} \\
\text{E} &::= \text{E}_1[\text{E}_2] \quad \begin{cases} &\text{E.type} \leftarrow \text{if E}_2.\text{type} = \text{integer} \text{ and} \\
&\text{E}_1.\text{type} = \text{array(s,t)} \text{ then t} \\
&\text{else typeError} \end{cases} \\
\text{E} &::= \text{E}_1 \uparrow \quad \begin{cases} &\text{E.type} \leftarrow \text{if E}_1.\text{type} = \text{pointer} \\
&\text{then t else typeError} \end{cases}
\end{align*}
\]
Type checking statements

Statements do not typically have values, therefore we assign them the type \textit{void}. If an error is detected within the statement, it gets type \textit{typeError}.

\begin{align*}
S & := \text{id} \leftarrow E \quad \{ \begin{array}{l}
S.type \leftarrow \text{id.type} = E.type \\
\quad \text{then} \text{ void} \\
\quad \text{else} \text{ typeError} \}
\end{array} \\
S & := \text{if} \ E \ \text{then} \ S_1 \quad \{ \begin{array}{l}
S.type \leftarrow \text{E.type} = \text{boolean} \\
\quad \text{then} \ S_1.type \\
\quad \text{else} \text{ typeError} \}
\end{array} \\
S & := \text{while} \ E \ \text{do} \ S_1 \quad \{ \begin{array}{l}
S.type \leftarrow \text{E.type} = \text{boolean} \\
\quad \text{then} \ S_1.type \\
\quad \text{else} \text{ typeError} \}
\end{array}
\end{align*}
Is This Really “Ad-hoc”?

Relationship between practice and attribute grammars

Similarities

- Both rules & actions associated with productions
- Application order determined by tools, not author
- (Somewhat) abstract names for symbols

Differences

- Actions applied as a unit; not true for AG rules
- Anything goes in ad-hoc actions; AG rules are functional
- AG rules are higher level than ad-hoc actions