Syntax Directed Translation

Attribute grammar and translation schemes
Typical implementation of languages

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → Target Program

Program input → Results

Interpreters

Compilers
Syntax-directed translation

- Compilers translate language constructs
  - Need to keep track of relevant information
    - Attributes: relevant information associated with a construct

\[ e ::= n \mid e + e \mid e - e \mid e \ast e \mid e / e \]

Attributes for expressions:
  - type of value: int, float, double, char, string,…
  - type of construct: variable, constant, operations, …

Attributes for constants: values
Attributes for variables: name, scope
Attributes for operations: arity, operands, operator,…
Syntax directed definition

- Associate a set of attributes with each grammar symbol
- Associate a set of semantic rules with each production
  - Specify how to compute attribute values of symbols

\[ e ::= n \mid e + e \mid e - e \mid e \ast e \mid e / e \]

Parse tree for 5 + 15 * 20:

Annotated parse tree:
**Synthesized attribute definition**

- An attribute is synthesized if
  - The attribute value of parent is determined from attribute values of children in the parse tree

\[
e ::= n \mid e + e \mid e - e \mid e \times e \mid e / e
\]

<table>
<thead>
<tr>
<th>production</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e ::= n )</td>
<td>( e.val = n.val )</td>
</tr>
<tr>
<td>( e ::= e1 + e2 )</td>
<td>( e.val = e1.val [+] e2.val )</td>
</tr>
<tr>
<td>( e ::= e1 - e2 )</td>
<td>( e.val = e1.val [-] e2.val )</td>
</tr>
<tr>
<td>( e ::= e1 \times e2 )</td>
<td>( e.val = e1.val [*] e2.val )</td>
</tr>
<tr>
<td>( e ::= e1 / e2 )</td>
<td>( e.val = e1.val [/] e2.val )</td>
</tr>
</tbody>
</table>
Inherited attribute definition

- An attribute is inherited if
  - The attribute value of a parse-tree node is determined from attribute values of its parent and siblings

```
D ::= T L
T ::= int | real
L ::= L , id | id
```

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>D ::= T L</td>
<td>L.in := T.type</td>
</tr>
<tr>
<td>T ::= int</td>
<td>T.Type := integer</td>
</tr>
<tr>
<td>T ::= real</td>
<td>T.type := real</td>
</tr>
<tr>
<td>L ::= L1 , id</td>
<td>L1.in := L.in Addtype(id.entry,L.in)</td>
</tr>
<tr>
<td>L ::= id</td>
<td>Addtype(id.entry,L.in)</td>
</tr>
</tbody>
</table>
Synthesized and inherited attributes

- Sometimes both synthesized and inherited attributes are required to evaluate necessary information.

```
e ::= n e'
e' ::= +ee' | *ee' | ε
```

<table>
<thead>
<tr>
<th>production</th>
<th>Semantic rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>e ::= n e'</td>
<td>e'.inh=n.val; e.val = e'.syn</td>
</tr>
<tr>
<td>e' ::= + e e'</td>
<td>e’1.inh = e’.inh [+] e.val; e’.syn = e’1.syn</td>
</tr>
<tr>
<td>e' ::= * e e'</td>
<td>e’1.inh = e’.inh [*] e.val; e’.syn = e’1.syn</td>
</tr>
<tr>
<td>e' ::= ε</td>
<td>e’.syn = e’.inh</td>
</tr>
</tbody>
</table>
Dependences in semantic evaluation

- If value of attribute b depends on attribute c,
  - Semantic rule for b must be evaluated after semantic rule for c
  - There is a dependence from c to b

**Annotated parse tree:**

```
D
  T.type=real
    real
      L1.in=real
        id1
      L2.in=real
        id2
      L3.in=real
        id3
```

**Dependency graph:**

```
real → T.type → L3.in → L3.addentry
      
L2.in → L2.addentry
      
L1.in
      
L1.addentry
      
id3.entry

id2.entry

id1.entry
```
Evaluation order of semantics

- Topological order of the dependence graph
  - Edges go from nodes earlier in the ordering to later nodes
  - No cycles are allowed in dependence graph

```
Input string  Parse tree  Dependency graph  Evaluation order for Semantic rules
```

```
real  T.type  L3.in  L3.addentry  id3.entry  id2.entry  id1.entry
  4    5      6       7         8        9         10
```

```
L1.in  L1.addentry  L2.in  L2.addentry
  9     10       7      8
```
Evaluation of semantic rules

- **Parse-tree methods (compile time)**
  - Build a parse tree for each input
  - Build a dependency graph from the parse tree
  - Obtain evaluation order from a topological order of the dependency graph

- **Rule-based methods (compiler-construction time)**
  - Predetermine the order of attribute evaluation for each production

- **Oblivious methods**
  - Evaluation order is independent of semantic rules
  - Evaluation order forced by parsing methods
  - Restrictive in acceptable attribute definitions
Bottom-up evaluation of attributes

- S-attributed definitions
  - Syntax-directed definitions with only synthesized attributes
  - Can be evaluated through post-order traversal of parse tree
- Synthesized attributes and bottom-up parsing
  - Keep attribute values of grammar symbols in stack
  - Evaluate attribute values at each reduction
- In top-down parsing, the return value of each parsing routine

Configuration of LR parser:

\[(s_0X_1s_1X_2s_2...X_m s_m, a_ia_{i+1}...a_n$, $v_1v_2...v_m)\]

states \hspace{2cm} inputs \hspace{2cm} values

Right-sentential form: \(X_1X_2...X_m a_ia_{i+1}...a_n\$

Automata states: \(s_0s_1s_2...s_m\)

Grammar symbols in stack: \(X_1X_2...X_m\)

Synthesized attribute values of \(X_i \rightarrow v_i\)
Implementing S-attributed definitions

Implementation of a desk calculator with an LR parser
(when a number is shifted onto symbol stack,
its value is shifted onto val stack)

<table>
<thead>
<tr>
<th>production</th>
<th>Code fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>E' ::= E</td>
<td>Print(val[top])</td>
</tr>
<tr>
<td>E ::= E1 + T</td>
<td>v=val[top-2]+val[top]; top-=2; val[top]=v;</td>
</tr>
<tr>
<td>E ::= T</td>
<td></td>
</tr>
<tr>
<td>T ::= T1 * F</td>
<td>v=val[top-2]*val[top]; top-=2; val[top]=v;</td>
</tr>
<tr>
<td>T ::= F</td>
<td></td>
</tr>
<tr>
<td>F ::= (E)</td>
<td>v=val[top-1]; top-=2; val[top]=v</td>
</tr>
<tr>
<td>F ::= n</td>
<td></td>
</tr>
</tbody>
</table>
L-attributed definitions

- A syntax-directed definition is L-attributed if each inherited attribute of $X_j$, $1 \leq j \leq n$, on the right side of $A::=X_1X_2\ldots X_n$, depends only on
  - the attributes of $X_1,X_2,\ldots,X_{j-1}$ to the left of $X_j$ in the production
  - the inherited attributes of $A$

<table>
<thead>
<tr>
<th>L-attributed definition</th>
<th>Non L-attributed definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Production</strong></td>
<td><strong>Semantic rules</strong></td>
</tr>
<tr>
<td>$D::=T L$</td>
<td>$L.in:=T.type$</td>
</tr>
<tr>
<td>$T ::= \textbf{int}$</td>
<td>$T.Type:=\text{integer}$</td>
</tr>
<tr>
<td>$T ::= \textbf{real}$</td>
<td>$T.type:=\text{real}$</td>
</tr>
<tr>
<td>$L ::= L_1 ,id$</td>
<td>$L_1.in := L.in$</td>
</tr>
<tr>
<td></td>
<td>Addtype(id.entry,L.in)</td>
</tr>
<tr>
<td>$L ::= id$</td>
<td>Addtype(id.entry,L.in)</td>
</tr>
</tbody>
</table>
Translation schemes

- A translation scheme is a CFG where
  - Attributes are associated with grammar symbols and
  - Semantic actions are inserted within right sides of productions
- Notation for specifying translation during parsing

Translation scheme:

```
E ::= T R
R ::= ‘+’ T {print(‘+’)} R1 | ε
T ::= num {print(num.val)}
```

Parse tree for 9+5 with actions

Treat actions as though they are terminal symbols.
Designing translation schemes

How to compute attribute values at each production?

<table>
<thead>
<tr>
<th>D::=T L</th>
<th>L.in:=T.type</th>
</tr>
</thead>
<tbody>
<tr>
<td>T::= int</td>
<td>T.Type:=integer</td>
</tr>
<tr>
<td>T::=real</td>
<td>T.type:=real</td>
</tr>
<tr>
<td>L::= id, L1</td>
<td>L1.in := L.in; Addtype(id.entry,L.in)</td>
</tr>
<tr>
<td>L::=id</td>
<td>Addtype(id.entry,L.in)</td>
</tr>
</tbody>
</table>

- Every attribute value must be available when referenced
  - S-attribute of left-hand symbol computed at end of production
  - I-attribute of right-hand symbol computed before the symbol
  - S-attribute of right-hand symbol referenced after the symbol

```plaintext
D::=T { L.in:=T.type} L
T::= int {T.Type:=integer}
T::=real {T.type:=real}
L::= id, {Addtype(id.entry,L.in)} {L1.in := L.in} L1
L::=id {Addtype(id.entry,L.in)}
```
Top-down translation

```c
void parseD()
{
    Type t = parseT();
    parseL(t);
}

Type parseT
{
    switch (currentToken()) {
        case INT: return TYPE_INT;
        case REAL: return TYPE_REAL;
    }
}

void parseL(Type in)
{
    SymEntry e = parseID();
    AddType(e, in);
    if (currentToken() == COMMA) {
        parseTerminal(COMMA);
        parseL(in)
    }
}
```
Top-down translation

- For each non-terminal A, construct a function that
  - Has a formal parameter for each inherited attribute of A
  - Returns the values of the synthesized attributes of A
- The code associated with each production does the following
  - Save the s-attribute of each token X into a variable X.x
  - Generate an assignment B.s=parseB(B.i1,B.i2,...,B.ik) for each non-terminal B, where B.i1,...,B.ik are values for the L-attributes of B and B.s is a variable to store s-attributes of B.
  - Copy the code for each action, replacing references to attributes by the corresponding variables
Bottom-up translation in Yacc

D::=T { L.in:=T.type} L
T::= int  {T.Type:=integer}
T::=real  { T.type:=real}
L::= {L1.in := L.in} L1,id {Addtype(id.entry,L.in) }
L::=id  {Addtype(id.entry,L.in)}

D : T  {$$ = $1; } L
T : INT { $$ = integer; } | REAL { $$ = real; } 
L : L COMMA ID { Addtype($3, $0); } 
   | ID { Addtype($1,$0); }