

Lecture Outline

1. The "dangling-else" ambiguity
2. Syntax-directed definitions and translation schemes
3. Synthesized and inherited attributes
4. S-attributed SDDs
5. L-attributed SDDs
6. Reading

1. The "Dangling-Else" Ambiguity

- Consider the following simplified ambiguous grammar for if- and if-else statements:

$$S \rightarrow \hat{a} \mid S$$

$$S \rightarrow \hat{a} \mid i S e S \mid i S \mid a$$

- Here the symbol i stands for `if expr then`, the symbol e stands for `else`, and the symbol a stands for all other productions. We have also added an augmenting production $S \rightarrow \hat{a} \mid S$.
- The canonical collections of sets of LR(0) items for this grammar is as follows:

$I_0: S \rightarrow \hat{a} \mid .S$	$I_3: S \rightarrow \hat{a} \mid a.$
$S \rightarrow \hat{a} \mid .iSeS$	
$S \rightarrow \hat{a} \mid .iS$	$I_4: S \rightarrow \hat{a} \mid iS.eS$
$S \rightarrow \hat{a} \mid .a$	$S \rightarrow \hat{a} \mid iS.$
$I_1: S \rightarrow \hat{a} \mid S.$	$I_5: S \rightarrow \hat{a} \mid iSe.S$
	$S \rightarrow \hat{a} \mid .iSeS$
$I_2: S \rightarrow \hat{a} \mid i.SeS$	$S \rightarrow \hat{a} \mid .iS$
$S \rightarrow \hat{a} \mid i.S$	$S \rightarrow \hat{a} \mid .a$
$S \rightarrow \hat{a} \mid .iSeS$	
$S \rightarrow \hat{a} \mid .iS$	$I_6: S \rightarrow \hat{a} \mid iSeS.$
$S \rightarrow \hat{a} \mid .a$	

- The set of items I_4 gives rise to a shift/reduce conflict. The item $S \rightarrow \hat{a} \mid iS.eS$ calls for a shift on e and since $\text{FOLLOW}(S) = \{e, \$\}$, the item $S \rightarrow \hat{a} \mid iS.$ calls for a reduction by production $S \rightarrow \hat{a} \mid iS.$ on e . To associate each e with the closest unused `if`, we should resolve the conflict in favor of shift to state 5.
- See Section 4.8.2 of ALSU for a more detailed discussion.

2. Syntax-Directed Definitions and Translation Schemes

- The syntax analyzer translates its input token stream into an intermediate language representation of the source program, usually an abstract syntax tree (AST).
- A syntax-directed definition can be used to specify this translation.
- A syntax-directed definition (SDD) is a context-free grammar with attributes attached to grammar symbols and semantic rules attached to the productions.
- The semantic rules define values for attributes associated with the symbols of the productions. These values can be computed by creating a parse tree for the input and then making a sequence of passes over the parse tree, evaluating some or all of the rules on each pass. SDDs are useful for specifying translations.
- A syntax-directed translation scheme (SDTS) is a context-free grammar with program fragments, called semantic actions, embedded within production bodies. SDTSs are useful for implementing syntax-directed definitions.

3. Synthesized and Inherited Attributes

- Attributes are values computed at the nodes of a parse tree.
- Synthesized attributes* are values that are computed at a node N in a parse tree from attribute values of the children of N and perhaps N itself. Synthesized attributes can be easily computed by a shift-reduce parser that keeps the values of the attributes on the parsing stack. See Sect. 5.4.2 of ALSU.
- An SDD is *S-attributed* if every attribute is synthesized. S-attributed SDDs are useful for bottom-up parsing.
- Inherited attributes* are values that are computed at a node N in a parse tree from attribute values of the parent of N , the siblings of N , and N itself.
- An SDD is *L-attributed* if every attribute is either synthesized or inherited from the parent or from the left. L-attributed SDDs are useful for top-down parsing. See Sect. 5.2.4 of ALSU for details.

4. Examples of S-Attributed SDDs

- **Example 1.** Here is an S-attributed SDD translating signed bit strings into decimal numbers. The attributes, `BNum.val`, `Sign.val`, `List.val`, and `Bit.val`, are all synthesized attributes that represent integers.

```
BNum  $\hat{+}$  Sign List      { BNum.val = Sign.val  $\tilde{-}$  List.val }
Sign  $\hat{+}$  ' +           { Sign.val = +1 }
Sign  $\hat{+}$  ' -           { Sign.val = -1 }
List  $\hat{+}$  List1 Bit     { List.val = 2  $\tilde{-}$  List1.val + Bit.val }
List  $\hat{+}$  Bit           { List.val = Bit.val }
Bit   $\hat{+}$  ' 0           { Bit.val = 0 }
Bit   $\hat{+}$  ' 1           { Bit.val = 1 }
```

- **Example 2.** Here are Yacc translation rules implementing the SDD above for translating signed bit strings into decimal numbers. The identifiers `$$`, `$1`, `$2` and so on in Yacc actions are synthesized attributes.

```
BNum : Sign List      { $$ = $1 * $2; }
    ;
Sign : '+'            { $$ = +1; }
    | '-'            { $$ = -1; }
    ;
List : List Bit       { $$ = 2*$1 + $2; }
    | Bit
    ;
Bit  : '0'            { $$ = 0; }
    | '1'            { $$ = 1; }
    ;
```

- **Example 3.** Here is an S-attributed SDD based on an SLR(1) grammar that translates arithmetic expressions into ASTs. `E` has the synthesized attributed `E.node` and `T` the synthesized attribute `T.node`. `E.node` and `T.node` point to a node in the AST. The function `Node(op, left, right)` returns a pointer to a node with three fields: the first labeled `op`, the second a pointer to a left subtree, and the third a pointer to a right subtree. The function `Leaf(op, value)` returns a pointer to a node with two fields: the first labeled `op`, the second the value of the token. See Example 5.11 in ALSU.

```
E  $\hat{+}$  E1 + T      { E.node = Node('+', E1.node, T.node); }
E  $\hat{+}$  T            { E.node = T.node; }
T  $\hat{+}$  ( E )        { T.node = E.node; }
T  $\hat{+}$  id           { T.node = Leaf(id, id.entry); }
```

5. Example of an L-Attributed SDD

- **Example 4.** Here is an L-attributed SDD based on an LL(1) grammar for translating arithmetic expressions into ASTs. See Example 5.12 in ALSU.

```
E  $\hat{+}$  T A          { E.node = A.s;
                  A.i = T.node; }
A  $\hat{+}$  + T A1      { A1.i = Node('+', A.i, T.node);
                  A.s = A1.s; }
A  $\hat{+}$   $\hat{\mu}$            { A.s = A.i; }
T  $\hat{+}$  ( E )        { T.node = E.node; }
T  $\hat{+}$  id           { T.node = Leaf(id, id.entry); }
```

6. Practice Problems

1. Using Yacc, implement a syntax-directed translator that translates sequences of postfix Polish expressions into infix notation. For example, your translator should map $345+*$ into $3*(4+5)$.
2. Optimize your translator so it doesn't generate any redundant parentheses. For example, your translator should still map $345+*$ into $3*(4+5)$ but it should map $345*+$ into $3+4*5$.

7. Reading

- ALSU, Sects. 5.1-5.4

aho@cs.columbia.edu