Motivation: parser as a translator

Syntax-directed translation

stream of tokens  parser  ASTs, or assembly code

syntax + translation rules (typically hardcoded in the parser)

Outline

- Syntax directed translation: specification
  - translate parse tree to its value, or to an AST
  - typecheck the parse tree

- Syntax-directed translation: implementation
  - during LR parsing
  - during LL parsing

To translate an input string:

1. Build the parse tree.
2. Working bottom-up
   - Use the translation rules to compute the translation of each nonterminal in the tree

Result: the translation of the string is the translation of the parse tree’s root nonterminal.

Why bottom up?
- A nonterminal’s value may depend on the value of the symbols on the right-hand side,
- So translate a non-terminal node only after children translations are available.

Example 1: arith expr to its value

Syntax-directed translation:

the CFG translation rules

E \rightarrow E + T \quad E\text{.trans} = E\text{.trans} + T\text{.trans}
E \rightarrow T \quad E\text{.trans} = T\text{.trans}
T \rightarrow T * F \quad T\text{.trans} = T\text{.trans} * F\text{.trans}
T \rightarrow F \quad T\text{.trans} = F\text{.trans}
F \rightarrow \text{int} \quad F\text{.trans} = \text{int value}
F \rightarrow ( E ) \quad F\text{.trans} = E\text{.trans}
Example 1 (cont)
Input: \(2 \times (4 + 5)\)

Annotated Parse Tree

Example 2: Compute the type of an expression

\[
E \rightarrow E + E \quad \text{if } ((E_2 \text{ trans} == \text{INT}) \text{ and } (E_3 \text{ trans} == \text{INT})) \\
\quad \text{then } E_1 \text{ trans} = \text{INT} \\
\quad \text{else } E_1 \text{ trans} = \text{ERROR}
\]

\[
E \rightarrow E \text{ and } E \quad \text{if } ((E_2 \text{ trans} == \text{BOOL}) \text{ and } (E_3 \text{ trans} == \text{BOOL})) \\
\text{then } E_1 \text{ trans} = \text{BOOL} \\
\quad \text{else } E_1 \text{ trans} = \text{ERROR}
\]

\[
E \rightarrow E == E \quad \text{if } ((E_2 \text{ trans} == E_3 \text{ trans}) \text{ and } (E_2 \text{ trans} != \text{ERROR})) \\
\text{then } E_1 \text{ trans} = \text{BOOL} \\
\quad \text{else } E_1 \text{ trans} = \text{ERROR}
\]

\[
E \rightarrow \text{true} \quad E \text{ trans} = \text{BOOL}
\]

\[
E \rightarrow \text{false} \quad E \text{ trans} = \text{BOOL}
\]

\[
E \rightarrow \text{int} \quad E \text{ trans} = \text{INT}
\]

\[
E \rightarrow (E) \quad E_1 \text{ trans} = E_2 \text{ trans}
\]

Example 2 (cont)

- Input: \((2 + 2) == 4\)

1. parse tree:

2. annotation:

TEST YOURSELF #1

• A CFG for the language of binary numbers:

\[
B \rightarrow 0 \\
\quad \rightarrow 1 \\
\quad \rightarrow B 0 \\
\quad \rightarrow B 1
\]

• Define a syntax-directed translation so that the translation of a binary number is its base-10 value.

• Draw the parse tree for 1001 and annotate each nonterminal with its translation.

Building Abstract Syntax Trees

- Examples so far, streams of tokens translated into
  - integer values, or
  - types

- Translating into ASTs is not very different

AST vs Parse Tree

- AST is condensed form of a parse tree
  - operators appear at internal nodes, not at leaves.
  - "Chains" of single productions are collapsed.
  - Lists are "flattened".
  - Syntactic details are omitted
    - e.g., parentheses, commas, semi-colons

- AST is a better structure for later compiler stages
  - omits details having to do with the source language,
  - only contains information about the essential structure of the program.
Example: 2 * (4 + 5)  

Parse tree vs AST

```
        E
       /\  
      *  / 
     /   /  
   T   2 +
  /\   /  
 F ( E )
     /\   
    E   T
     /\   
    F   int(5)
```

AST-building translation rules

```
E₁ → E₂ + T  E₁.trans = new PlusNode(E₂.trans, T.trans)
E → T        E.trans = T.trans
T₁ → T₂ * F  T₁.trans = new TimesNode(T₂.trans, F.trans)
T → F        T.trans = F.trans
F → int      F.trans = new IntLitNode(int.value)
F → ( E )    F.trans = E.trans
```

TEST YOURSELF #2

- Illustrate the syntax-directed translation defined above by
  - drawing the parse tree for 2 * 3 + 4,
  - annotating the parse tree with its translation
  - i.e., each nonterminal X in the parse tree will have a pointer to the root of the AST subtree that is the translation of X.

Semantic actions during parsing

- when shifting
  - push the value of the terminal on the sem. stack
- when reducing
  - pop k values from the sem. stack, where k is the number of symbols on production's RHS
  - push the production's value on the sem. stack

Syntax-Directed Translation and LR Parsing

- add semantic stack,
  - parallel to the parsing stack:
    - each symbol (terminal or non-terminal) on the parsing stack stores its value on the semantic stack
    - holds terminals' attributes, and
    - holds nonterminals' translations
  - when the parse is finished, the semantic stack will hold just one value:
    - the translation of the root non-terminal (which is the translation of the whole input).

An LR example

Grammar + translation rules:
```
E₁ → E₂ + ( E₃ )  E₁.trans = E₂.trans + E₃.trans
E₁ → int          E₁.trans = int.trans
```

Input:
```
2 * (3) * (4)
```
Shift-Reduce Example with evaluations

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Semantic Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>int + (int) + (int)$</td>
<td>shift</td>
</tr>
</tbody>
</table>

Shift-Reduce Example with evaluations

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Semantic Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>int + (int) + (int)$</td>
<td>shift</td>
</tr>
<tr>
<td>int + (int) + (int)$</td>
<td>red. E - int 2</td>
</tr>
<tr>
<td>E + (int) + (int)$</td>
<td>shift 3 times 2</td>
</tr>
</tbody>
</table>

Shift-Reduce Example with evaluations

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Semantic Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>int + (int) + (int)$</td>
<td>shift</td>
</tr>
<tr>
<td>int + (int) + (int)$</td>
<td>red. E - int 2</td>
</tr>
<tr>
<td>E + (int) + (int)$</td>
<td>shift 3 times 2</td>
</tr>
<tr>
<td>E + (int) + (int)$</td>
<td>red. E - int 2 '+' '(' 3</td>
</tr>
</tbody>
</table>

Shift-Reduce Example with evaluations

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Semantic Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>int + (int) + (int)$</td>
<td>shift</td>
</tr>
<tr>
<td>int + (int) + (int)$</td>
<td>red. E - int 2</td>
</tr>
<tr>
<td>E + (int) + (int)$</td>
<td>shift 3 times 2</td>
</tr>
<tr>
<td>E + (int) + (int)$</td>
<td>red. E - int 2 '+' '(' 3</td>
</tr>
<tr>
<td>E + (E) + (int)$</td>
<td>shift 2 '+' '(' 3</td>
</tr>
</tbody>
</table>

Shift-Reduce Example with evaluations

<table>
<thead>
<tr>
<th>Parsing Stack</th>
<th>Semantic Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>int + (int) + (int)$</td>
<td>shift</td>
</tr>
<tr>
<td>int + (int) + (int)$</td>
<td>red. E - int 2</td>
</tr>
<tr>
<td>E + (int) + (int)$</td>
<td>shift 3 times 2</td>
</tr>
<tr>
<td>E + (int) + (int)$</td>
<td>red. E - int 2 '+' '(' 3</td>
</tr>
<tr>
<td>E + (E) + (int)$</td>
<td>shift 2 '+' '(' 3</td>
</tr>
<tr>
<td>E + (E) + (int)$</td>
<td>red. E - E + (E) 2 '+' '(' 3</td>
</tr>
</tbody>
</table>
### Shift-Reduce Example with evaluations

<table>
<thead>
<tr>
<th>✦ int • (int) • (int)$</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>int • (int) • (int)$</td>
<td>red. E - int</td>
</tr>
<tr>
<td>E • (int) • (int)$</td>
<td>shift 3 times</td>
</tr>
<tr>
<td>E • (int) • (int)$</td>
<td>red. E - int</td>
</tr>
<tr>
<td>E + (int) • (int)$</td>
<td>red. E - E • (E)</td>
</tr>
<tr>
<td>E • (int)$</td>
<td>shift 3 times</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>✦ int • (int) • (int)$</th>
<th>shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>int • (int) • (int)$</td>
<td>red. E - int</td>
</tr>
<tr>
<td>E • (int) • (int)$</td>
<td>shift 3 times</td>
</tr>
<tr>
<td>E • (int) • (int)$</td>
<td>red. E - int</td>
</tr>
<tr>
<td>E • (int) • (int)$</td>
<td>shift 3 times 2</td>
</tr>
<tr>
<td>E + (int) • (int)$</td>
<td>red. E - E • (E)</td>
</tr>
<tr>
<td>E • (int)$</td>
<td>shift 3 times</td>
</tr>
</tbody>
</table>

### Syntax-Directed Translation and LL Parsing

- not obvious how to do this, since
  - predictive parser builds the parse tree top-down, syntax-directed translation is computed bottom-up.
  - could build the parse tree (inefficient!)
  - Instead, the parsing stack will also contain actions
    - these actions will be delayed: to be executed when popped from the stack
  - To simplify the presentation (and to show you a different style of translation), assume:
    - only non-terminals’ values will be placed on the sem. stack
How does semantic stack work?

- How to push/pop onto/off the semantic stack?
  - add actions to the grammar rules.
- The action for one rule must:
  - Pop the translations of all rhs nonterminals.
  - Compute and push the translation of the lhs nonterminal.
- Actions are represented by action numbers,
  - action numbers become part of the rhs of the grammar rules.
  - action numbers pushed onto the (normal) stack along with the
    terminal and nonterminal symbols.
  - when an action number is the top-of-stack symbol,
    it is popped and the action is carried out.

Keep in mind

- action for \( X \to Y_1 Y_2 \ldots Y_n \) is pushed onto the
  (normal) stack when the derivation step
  \( X \to Y_1 Y_2 \ldots Y_n \) is made, but
- the action is performed only after complete
derivations for all of the \( Y \)'s have been
carried out.

Example: Counting Parentheses

\[
E_1 \to \varepsilon \quad E_1.\text{trans} = 0 \\
( E_2 ) \quad E_1.\text{trans} = E_2.\text{trans} + 1 \\
[ E_2 ] \quad E_1.\text{trans} = E_2.\text{trans}
\]

Example: Step 1

replace the translation rules with translation actions.
- Each action must:
  - Pop the nonterminals’ translations from the semantic stack.
  - Compute and push the lhs nonterminal’s translation.
- Here are the translation actions:

  \[
  \begin{align*}
  &E \to \varepsilon &\text{push(0);} \\
  &\to (E) &\text{exp2Trans} = \text{pop(); push( exp2Trans + 1 );} \\
  &\to [E] &\text{exp2Trans} = \text{pop(); push( exp2Trans );}
  \end{align*}
  \]

Example: Step 2

each action is represented by a unique action number,
- the action numbers become part of the grammar rules:

  \[
  \begin{align*}
  &E \to \varepsilon &\text{#1} \\
  &\to (E) &\text{#2} \\
  &\to [E] &\text{#3}
  \end{align*}
  \]

  #1: push(0);
  #2: exp2Trans = pop(); push( exp2Trans + 1 );
  #3: exp2Trans = pop(); push( exp2Trans );

Example: Example

<table>
<thead>
<tr>
<th>action</th>
<th>stack</th>
<th>semantic stack</th>
<th>input so far</th>
</tr>
</thead>
<tbody>
<tr>
<td>( { )</td>
<td>E EOF</td>
<td>replace E with ( E ) #2</td>
<td>{</td>
</tr>
<tr>
<td>( { )</td>
<td>(E) #2 EOF</td>
<td>terminal</td>
<td>(E) #2 EOF</td>
</tr>
<tr>
<td>( { )</td>
<td>E #2 EOF</td>
<td>replace E with [ E ]</td>
<td>[E] #2 EOF</td>
</tr>
<tr>
<td>( { )</td>
<td>(E) #2 EOF</td>
<td>terminal</td>
<td>(E) #2 EOF</td>
</tr>
<tr>
<td>({)</td>
<td>E #2 EOF</td>
<td>replace E with [ E ]</td>
<td>(E) #2 EOF</td>
</tr>
<tr>
<td>({)</td>
<td>} #1 EOF</td>
<td>pop action, do action</td>
<td>} #1 EOF</td>
</tr>
<tr>
<td>({)</td>
<td>} #2 EOF</td>
<td>0</td>
<td>} #2 EOF</td>
</tr>
<tr>
<td>({)</td>
<td>EOF</td>
<td>empty stack: input accepted!</td>
<td>EOF</td>
</tr>
<tr>
<td>({)</td>
<td>#2 EOF</td>
<td>0</td>
<td>translation of input + 1</td>
</tr>
<tr>
<td>({)</td>
<td>EOF</td>
<td>1</td>
<td>terminal</td>
</tr>
</tbody>
</table>
What if the rhs has >1 nonterminal?

- pop multiple values from the semantic stack:
  - **CFG Rule**: 
    \[
    \text{methodBody} \rightarrow \{ \text{varDecls} \text{ stmts} \}
    \]
  - **Translation Rule**: 
    \[
    \text{methodBody}.\text{trans} = \text{varDecls}.\text{trans} \times \text{stmts}.\text{trans}
    \]
  - **Translation Action**: 
    \[
    \text{stmtsTrans} = \text{pop}(); \text{declsTrans} = \text{pop}();
    \text{push( stmtsTrans + declsTrans )};
    \]
  - **CFG rule with Action**: 
    \[
    \text{methodBody} \rightarrow \{ \text{varDecls} \text{ stmts} \} \#1
    \]
    \[
    \#1: \text{stmtsTrans} = \text{pop}(); \text{declsTrans} = \text{pop}();
    \text{push( stmtsTrans + declsTrans )};
    \]

Terminals

- **Simplification**:
  - we assumed that each rhs contains at most one terminal
- **How to push the value of a terminal**:
  - a terminal’s value is available only when the terminal is the “current token”.
  - **put action before the terminal**
    - **CFG Rule**: 
      \[
      F \rightarrow \text{int}
      \]
    - **Translation Rule**: 
      \[
      F.\text{trans} = \text{int.value}
      \]
    - **Translation Action**: 
      \[
      \text{push( int.value )}
      \]
    - **CFG rule with Action**: 
      \[
      F \rightarrow \#1 \text{ int} // \text{action BEFORE terminal}
      \]
    - **push( currToken.value )

Handling non-LL(1) grammars

- Recall that to do LL(1) parsing
  - non-LL(1) grammars must be transformed
    - e.g., left-recursion elimination
    - the resulting grammar does not reflect the underlying structure of the program
    \[
    \text{E} \rightarrow \text{E} + \text{T}
    \]
    \[
    \text{E} \rightarrow \text{T} \text{E}'
    \]
    \[
    \text{E}' \rightarrow \varepsilon | + \text{T} \text{E}'
    \]
- **How to define syntax directed translation for such grammars?**

The solution is simple!

- **Treat actions as grammar symbols**
  - define syntax-directed translation on the original grammar:
    - define translation rules
    - convert them to actions that push/pop the semantic stack
    - incorporate the action numbers into the grammar rules
  - then convert the grammar to LL(1)
    - treat action numbers as regular grammar symbols

Example

non-LL(1):
\[
\begin{align*}
\text{E} & \rightarrow \text{E} + \text{T} \#1 | \text{T} \\
\text{T} & \rightarrow \text{T} * \text{F} \#2 | \text{F} \\
\text{F} & \rightarrow \#3 \text{ int}
\end{align*}
\]

\#1: \text{TTrans} = \text{pop}(); \text{ETrans} = \text{pop}(); \text{push(Etrans + TTrans)};

\#2: \text{FTrans} = \text{pop}(); \text{TTrans} = \text{pop}(); \text{push(Ttrans * FTrans)};

\#3: \text{push( int.value )};

after removing immediate left recursion:
\[
\begin{align*}
\text{E} & \rightarrow \text{E}' \\
\text{T} & \rightarrow \text{F} \#1 \\
\text{E}' & \rightarrow \text{E} | \varepsilon \\
\text{F} & \rightarrow \#3 \text{ int}
\end{align*}
\]

TEST YOURSELF #3

- For the following grammar, give
  - translation rules + translation actions,
  - a CFG with actions so that the translation of an input expression is the value of the expression.
  - Do not worry that the grammar is not LL(1).
  - then convert the grammar to LL(1)

\[
\begin{align*}
\text{E} & \rightarrow \text{E} + \text{T} | \text{E} - \text{T} | \text{T} \\
\text{T} & \rightarrow \text{T} * \text{F} | \text{T} / \text{F} | \text{F} \\
\text{F} & \rightarrow \text{int} | \left( \text{E} \right)
\end{align*}
\]