Syntax-Directed Translation

- Extending CFGs
- Grammar Annotation
- Parse Trees
- Abstract Syntax Trees (ASTs)

- Readings: Section 5.1, 5.2, 5.5, 5.6

Motivation: parser as a translator

Syntax-Directed Translation

stream of tokens → parser → ASTs, or assembly code

syntax + translation rules (typically hardcoded in the parser)
Mechanism of syntax-directed translation

- syntax-directed translation is done by extending the CFG
  - a translation rule is defined for each production

  given
  \[ X \rightarrow dABoC \]
  the translation of \( X \) is defined in terms of
  - translation of nonterminals \( A, B \)
  - values of attributes of terminals \( d, c \)
  - constants

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

\[
\begin{align*}
E \rightarrow E + T & \quad E_1.text = E_2.text + T.text \\
E \rightarrow T & \quad E.text = T.text \\
T \rightarrow T * F & \quad T_1.text = T_2.text * F.text \\
T \rightarrow F & \quad T.text = F.text \\
F \rightarrow \text{int} & \quad F.text = \text{int.value} \\
F \rightarrow ( E ) & \quad F.text = E.text
\end{align*}
\]

Example 1 (cont)

Input: 2 * (4 + 5)

Annotated Parse Tree
Example 2: Compute type of expr

E → E + E if (E₂.trans == INT) and (E₃.trans == INT)
   then E₁.trans = INT
   else E₁.trans = ERROR
E → E and E if (E₂.trans == BOOL) and (E₃.trans == BOOL)
   then E₁.trans = BOOL
   else E₁.trans = ERROR
E → E == E if (E₂.trans == E₃.trans) and (E₂.trans != ERROR)
   then E₁.trans = BOOL
   else E₁.trans = ERROR
E → true E.trans = BOOL
E → false E.trans = BOOL
E → int E.trans = INT
E → ( E ) E₁.trans = E₂.trans

Example 2 (cont)

- Input: (2 + 2) == 4
  
  1. parse tree:
  
  2. annotation:
Another Example

- A CFG for the language of binary numbers:
  \[ B \rightarrow 0 \]
  \[ \rightarrow 1 \]
  \[ \rightarrow B\ 0 \]
  \[ \rightarrow B\ 1 \]
- 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 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

---

**Ex: 2*(4+5) parse tree vs AST**

- **Parse Tree**
  - E
    - T
      - F
        - int (2)
  - T
    - F
      - int (4)
  - T
    - *
      - F
        - E
          - ( E )
            - T
              - *
                - T
                  - F
                    - int (5)
            - int (5)

- **AST**
  - *
    - 2
      - +
        - 4
          - 5
Definitions of AST nodes

```java
class ExpNode {
}

class IntLitNode extends ExpNode {
    public IntLitNode(int val) {...}
}

class PlusNode extends ExpNode {
    public PlusNode( ExpNode e1, ExpNode e2 ) {
        ...
    }
}

class TimesNode extends ExpNode {
    public TimesNode( ExpNode e1, ExpNode e2 ) {
        ...
    }
}
```

AST-building translation rules

```plaintext
E₁ \rightarrow E₂ + T  \quad E₁.trans =
                         \quad \text{new PlusNode}(E₂.trans, T.trans)

E \rightarrow T  \quad E.trans = T.trans

T₁ \rightarrow T₂ * F  \quad T₁.trans =
                         \quad \text{new TimesNode}(T₂.trans, F.trans)

T \rightarrow F  \quad T.trans = F.trans

F \rightarrow \text{int}  \quad F.trans = \text{new IntLitNode}(\text{int.value})

F \rightarrow ( E )  \quad F.trans = E.trans
```
Example

- Illustrate the syntax-directed translation defined previously by
  - drawing the parse tree for $2 + 3 \times 4$, and
  - 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 $X$

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, add a **semantic stack**:
  - holds nonterminals' translations
  - when the parse is finished, the semantic stack will hold just one value:
    - the translation of the root nonterminal (which is the translation of the whole input).
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 rhs of 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 $X \rightarrow Y_1 Y_2 \ldots Y_n$ is pushed onto the (normal) stack when the derivation step $X \rightarrow Y_1 Y_2 \ldots Y_n$ is made, but

- the action is performed only after complete derivations for all of the Y's have been carried out
Example: Counting Parentheses

E₁ → ε  E₁.trans = 0
→ ( E₂ )  E₁.trans = E₂.trans + 1
→ [ E₂ ]  E₁.trans = E₂.trans

Example: Step 1

- replace the translation rules with translation actions

  - Each action must:
    - Pop rhs nonterminals' translations from semantic stack
    - Compute and push the lhs nonterminal's translation

- Here are the translation actions:

  E → ε  push(0);
  → ( E )  exp2Trans = pop();
           push( exp2Trans + 1 );
  → [ E ]  exp2Trans = pop();
           push( exp2Trans );
Example: Step 2

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

\[ E \rightarrow \varepsilon \ #1 \]
\[ \rightarrow ( E ) \ #2 \]
\[ \rightarrow [ E ] \ #3 \]

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

Example: example

<table>
<thead>
<tr>
<th>input so far</th>
<th>stack</th>
<th>semantic stack</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>E EOF</td>
<td></td>
<td>pop, push &quot;( E ) #2&quot;</td>
</tr>
<tr>
<td>(</td>
<td>(E) #2 EOF</td>
<td>pop, scan</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>E) #2 EOF</td>
<td>pop, push &quot;[ E ]&quot;</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>[E] ) #2 EOF</td>
<td>pop, scan</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>[E] ) #2 EOF</td>
<td>pop, push ( \varepsilon \ #1</td>
<td></td>
</tr>
<tr>
<td>[</td>
<td>#1 ] ) #2 EOF</td>
<td>pop, do action</td>
<td></td>
</tr>
<tr>
<td>[</td>
<td>) #2 EOF</td>
<td>pop, scan</td>
<td></td>
</tr>
<tr>
<td>[</td>
<td>) #2 EOF</td>
<td>pop, scan</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>EOF</td>
<td>pop, do action</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>EOF</td>
<td>pop, scan</td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>EOF</td>
<td>empty stack: input accepted!</td>
<td>translation of input = 1</td>
</tr>
</tbody>
</table>
What if the rhs has >1 nonterminal?

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

---

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: \[ \text{push}(\text{int.value}) \]
  - CFG rule with Action:
    \[ F \rightarrow \#1 \text{int} \ // \text{action BEFORE terminal} \]
    \[ \text{#1: push}(\text{currToken.value}) \]
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
    \[ E \rightarrow E + T \]
    vs.
    \[ E \rightarrow T E' \]
    \[ E' \rightarrow \epsilon \mid + T E' \]

- 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):

\[ E \rightarrow E + T \ #1 \]
\[ \quad \rightarrow T \]
\[ T \rightarrow T * F \ #2 \]
\[ \quad \rightarrow F \]

#1: TTrans = pop(); ETrans = pop(); push Etrans + TTrans;
#2: FTrans = pop(); TTrans = pop(); push Ttrans * FTrans;

after removing immediate left recursion:

\[ E \rightarrow T \ E' \]
\[ E' \rightarrow + T \ #1 \ E' \]
\[ \quad \rightarrow \epsilon \]
\[ T \rightarrow F \ T' \]
\[ T' \rightarrow * F \ #2 \ T' \]
\[ \quad \rightarrow \epsilon \]

Example

- 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 (including actions)
  to LL(1)

\[
E \rightarrow E + T \mid E - T \mid T \\
T \rightarrow T * F \mid T / F \mid F \\
F \rightarrow \text{int} \mid (E)
\]