CS 536 Announcements for Wednesday, February 14, 2024

Programming Assignment 2
• due Tuesday, February 20

Last Time
• Makefiles
• ambiguous grammars
• grammars for expressions
  • precedence
  • associativity
• grammars for lists

Today
• syntax-directed translation
• intro to abstract syntax trees

Next Time
• implementing ASTs

Recall our expression grammar

Write an unambiguous grammar for integer expressions involving only addition, multiplication, and parentheses that correctly handles precedence and associativity.

\[

e \rightarrow \ e \ PLUS \ t \\
| \ e \\

t \rightarrow \ t \ TIMES \ f \\
| \ f \\

f \rightarrow \ INTLIT \\
| \ LPAREN \ e \ RPAREN
\]

Extend this grammar to add exponentiation (POW)

Add exponentiation (POW) to this grammar, with the correct precedence and associativity.

\[
f \rightarrow \ f \ POW \ f \\
| \ e \\

e \rightarrow \ INTLIT \\
| \ ( \ e \ )
\]
Overview of CFGs

CFGs for language definition
- the CFGs we've discussed can generate/define languages of valid strings

\[ \text{start by building parse tree} \quad \text{& end with some valid string } \text{WEL}(G) \]

CFGs for language recognition

\[ \text{start with string } w \quad \text{& end with yes/no answer depending on whether } \text{WEL}(G) \]

CFGs for parsing

\[ \text{start with string } w \quad \text{& end with parse tree for } w \text{ if } \text{WEL}(G) \]

- generally use AST instead of parse tree

\[ \text{need to translate sequence of tokens (W)} \]
Syntax-directed translation (SDT) = translating from a sequence of tokens into a sequence of actions/other form, based on underlying syntax

Could be: AST, value, type, etc.

To define a syntax-directed translation
Augment CFG with *translation rules* (at most 1 rule per production)
- define translation of LHS non-terminal as a function of
  - constants
  - translations of RHS non-terminals
  - values of tokens (terminals) on RHS

To translate a sequence of tokens using SDT
- build parse tree
- use translation rules to compute translation of each non-terminal in parse tree
  - bottom-up ← handle children of node before node
- translation of sequence of tokens is the translation of the parse tree's root non-terminal (i.e., start symbol)

The type of the translation can be anything: numeric, string, set, tree, ...

*Note: above is how to understand the translation, not how a compiler actually does it*
Example: grammar for language of binary numbers

**CFG**

\[
\begin{align*}
b & \rightarrow 0 \\
b & \rightarrow 1 \\
b & \rightarrow b \ 0 \\
b & \rightarrow b \ 1 \\
\end{align*}
\]

**translation rules**

\[
\begin{align*}
b \text{.trans} & = 0 \\
b \text{.trans} & = 1 \\
b_1 \text{.trans} & = b_2 \text{.trans} \times 2 \\
b_1 \text{.trans} & = b_2 \text{.trans} \times 2 + 1 \\
\end{align*}
\]

Example: input string is 10110

Parse tree:

Translation is 22

SDT to compute the decimal equivalent of a binary number
Example: grammar for language of variable declarations

CFG

- declList \rightarrow \varepsilon
  \mid \text{decl \ declList}
- decl \rightarrow \text{type \ ID ;}
- type \rightarrow \text{INT}
  \mid \text{BOOL}

Translation rules

\[
\begin{align*}
\text{declList}.trans &= \"\" \\
\text{declList}_1.trans &= \text{decl}.trans + \"\" + \text{declList}_2.trans \\
\text{decl}.trans &= \text{ID}.value
\end{align*}
\]

Write a syntax-directed translation for the CFG given above so that the translation of a sequence of tokens is a string containing the ID's that have been declared.

Example input
- int xx;
- bool bb;
- int yy;

Parse tree

Example output
- "bb xx yy"

(in any order)
Example: grammar for language of variable declarations

CFG

<table>
<thead>
<tr>
<th>Production</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declList → ε</td>
<td></td>
</tr>
<tr>
<td>declList → decl declList</td>
<td></td>
</tr>
<tr>
<td>decl → type ID ;</td>
<td></td>
</tr>
<tr>
<td>type → INT</td>
<td></td>
</tr>
<tr>
<td>type → BOOL</td>
<td></td>
</tr>
</tbody>
</table>

Translation rules

<table>
<thead>
<tr>
<th>Production</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>declList.trans = ''</td>
<td></td>
</tr>
<tr>
<td>declList.trans = decl.trans + declList2.trans</td>
<td></td>
</tr>
<tr>
<td>decl.trans = type.trans + ID.value *</td>
<td></td>
</tr>
<tr>
<td>type.trans = true</td>
<td></td>
</tr>
<tr>
<td>type.trans = false</td>
<td></td>
</tr>
</tbody>
</table>

Modify the previous syntax-directed translation so that only declarations of type \textit{int} are added to the output string.

\textbf{Example input}

\begin{align*}
\text{int } & xx ; \\
\text{bool } & bb ; \\
\text{int } & yy ; \\
\end{align*}

\textbf{Output}

\begin{align*}
\text{'' } & xx yy \text{''} \\
\text{(in any order)} & \\
\text{equiv to} & \\
\text{if (a)} & \\
\text{x=b} & \\
\text{else} & \\
\text{x=c} & \\
\end{align*}

\textbf{Note:}

1) different nonterm can have different types as their translation
2) translation rules can be conditional
SDT for parsing

Previous examples showed SDT process assigning different types to the translation

- translate tokenized stream to an integer value
- translate tokenized stream to a string

For parsing, we'll need to translate a tokenized stream to an abstract-syntax tree (AST)

Abstract syntax trees

**AST** = condensed form of parse tree

- operators at internal nodes (not leaves)
- chains of productions are collapsed
- list are flattened
- syntactic details are omitted

Example: paren
AST Example

CFG

expr → expr PLUS term
| term

term → term TIMES factor
| factor

factor → INTLIT
| LPAREN expr RPAREN

Parse tree

\[(5 + 2) \times 8\]