G22.2130-001

Compiler Construction

Lecture 3:
Syntax-Directed Translator (Cont'd)

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
From Last Time

Attributes
- With terminals and nonterminals
- Semantic rules with each production
- Semantic rules explain how to calculate head attribute from body of production

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{term} \\
& \mid \text{expr} - \text{term} \\
& \mid \text{term} \\
\text{digit} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*}
\]

\[
\begin{align*}
\text{expr}.t & = 95-2+ \\
\text{expr}.t & = 95- \\
\text{term}.t & = 2 \\
\text{expr}.t & = 9 - \text{term}.t \\
\text{term}.t & = 5 \\
\text{term}.t & = 2
\end{align*}
\]

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr \rightarrow expr + term</td>
<td>expr.t = expr.t</td>
</tr>
<tr>
<td>expr \rightarrow expr - term</td>
<td>expr.t = expr.t</td>
</tr>
<tr>
<td>expr \rightarrow term</td>
<td>expr.t = term.t</td>
</tr>
<tr>
<td>term \rightarrow 0</td>
<td>term.t = '0'</td>
</tr>
<tr>
<td>term \rightarrow 1</td>
<td>term.t = '1'</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>term \rightarrow 9</td>
<td>term.t = '9'</td>
</tr>
</tbody>
</table>
From Last Time

Translation schemes

– Instead of attributes add program fragment to production rules

– They are called semantic actions
Exercise 2.3.1: Construct a syntax-directed translation scheme that translates arithmetic expressions from infix notation into prefix notation in which an operator appears before its operands; e.g., $-xy$ is the prefix notation for $x - y$. Give annotated parse trees for the inputs $9-5+2$ and $9-5*2$. 
**Predictive Parsing**

\[
\text{stmt} \rightarrow \text{expr} \; ; \\
\mid \; \text{if} ( \text{expr} ) \; \text{stmt} \\
\mid \; \text{for} ( \text{optexpr} ; \text{optexpr} ; \text{optexpr} ) \\
\mid \; \text{other} \\
\]

\[
\text{optexpr} \rightarrow \epsilon \\
\mid \; \text{expr} \\
\]

```c
void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt(); break;
    case for:
        match(for); match('(');
        optexpr(); match(')'); optexpr(); match(')'); optexpr();
        match(')'); stmt(); break;
    case other;
        match(other); break;
    default:
        report("syntax error");
    }
}

void optexpr() {
    if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
    if ( lookahead == t ) lookahead = nextTerminal;
    else report("syntax error");
}
```
Each nonterminal becomes a procedure.

```c
void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(')'); optexpr(); match(')'); optexpr();
        match(')'); stmt(); break;
    case other;
        match(other); break;
    default:
        report("syntax error");
    }
}

void optexpr() {
    if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
    if ( lookahead == t ) lookahead = nextTerminal;
    else report("syntax error");
}
```
Predictive Parsing

Terminal is matched and lookahead advances.
The Evil in Predictive Parsing: Left Recursion

expr → expr + term

This can loop forever. Can you see why?

We can eliminate

\[ A \rightarrow A\alpha \mid \beta \]

As follows:

\[
\begin{align*}
A & \rightarrow \beta R \\
R & \rightarrow \alpha R \mid \epsilon
\end{align*}
\]
Let’s Build A Translator: Arithmetic Expressions to Postfix

expr → expr + term { print(‘+’) }
      | expr - term { print(‘-’) }
      | term

term → 0 { print(‘0’) }
      | 1 { print(‘1’) }
      | ... 
      | 9 { print(‘9’) }

Do you see any problems with this production?
Let's Build A Translator: Arithmetic Expressions to Postfix

\[
\begin{align*}
expr & \rightarrow \ expr + \ term \quad \{ \text{print('+' )} \} \\
& \quad | \quad \ expr - \ term \quad \{ \text{print('-') } \} \\
& \quad | \quad \ term \\
\term & \rightarrow \ 0 \quad \{ \text{print('0') } \} \\
& \quad | \quad 1 \quad \{ \text{print('1') } \} \\
& \quad | \quad \ldots \ \\
& \quad | \quad 9 \quad \{ \text{print('9') } \}
\end{align*}
\]

\[
A \rightarrow A\alpha \mid \beta \\
A \rightarrow \beta R \\
R \rightarrow \alpha R \mid \epsilon
\]

Can we apply the above rule here?
Let’s Build A Translator: Arithmetic Expressions to Postfix

\[
\begin{align*}
expr & \rightarrow \text{expr} + \text{term} \quad \{ \text{print(’+’) } \} \\
& \quad \mid \text{expr} - \text{term} \quad \{ \text{print(’-’) } \} \\
& \quad \mid \text{term} \\
\text{term} & \rightarrow 0 \quad \{ \text{print(’0’) } \} \\
& \quad \mid 1 \quad \{ \text{print(’1’) } \} \\
& \quad \mid \ldots \\
& \quad \mid 9 \quad \{ \text{print(’9’) } \}
\end{align*}
\]

\[
A \rightarrow A\alpha \mid A\beta \mid \gamma \\
A \rightarrow \gamma R \\
R \rightarrow \alpha R \mid \beta R \mid \epsilon
\]

Let's Build A Translator: Arithmetic Expressions to Postfix

expr → expr + term \{ print('+' ) \}
   | expr - term \{ print('-') \}
   | term

term → 0 \{ print('0') \}
    | 1 \{ print('1') \}
    | ... 
    | 9 \{ print('9') \}

A → Aα | Aβ | γ
A → γR
R → αR | βR | ε

A = expr
α = + term \{ print('+') \}
β = - term \{ print('-') \}
γ = term
Let's Build A Translator: Arithmetic Expressions to Postfix

expr → term rest

rest → + term { print('+' ) } rest
| - term { print('-') } rest
| ε

term → 0 { print('0') }
| 1 { print('1') }
| ... 
| 9 { print('9') }

A → Aα | Aβ | γ
A → γR
R → αR | βR | ε

Can you show the translation of 9-5+2 ?

Can we write now a pseudocode for it?
Lexical Analysis

• Reads characters from the input and groups them into tokens
• Sequence of characters that comprises a single token is called lexeme
• Lexical analyzer isolates the parser from lexemes
What Is A Token?

• It is a way of categorization
• In English it can be:
  noun, verb, adjective, ...
• In programming language it is:
  Identifier, keyword, integer, ...
• Parser relies on tokens distinctions
\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{term} \quad \{ \text{print}'+' \} \\
& \mid \text{expr} - \text{term} \quad \{ \text{print}'-' \} \\
& \mid \text{term} \\
\text{term} & \rightarrow \text{term} * \text{factor} \quad \{ \text{print}'*' \} \\
& \mid \text{term} / \text{factor} \quad \{ \text{print}'/' \} \\
& \mid \text{factor} \\
\text{factor} & \rightarrow ( \text{expr} ) \\
& \mid \text{num} \quad \{ \text{print}(\text{num}.\text{value}) \} \\
& \mid \text{id} \quad \{ \text{print}(\text{id}.\text{lexeme}) \}
\end{align*}
\]
Thanks to the lexical analyzer the parser can deal with identifier

Thanks to the lexical analyzer the parser can deal with any number
Issues in Lexical Analysis

• White spaces removal
• Comments removal
• Integer constants
• Recognizing keywords and identifiers
White Space Removal

Makes parser’s life much easier

```c
for (; ; peek = next input character ) {
    if ( peek is a blank or a tab ) do nothing;
    else if ( peek is a newline ) line = line+1;
    else break;
}
```
Reading Ahead

• Lexical analyzer may need to read several characters ahead
• Helps in decision making
• Fetching block of characters is more efficient than fetching a character at a time
• A buffer is needed
Integer Constants

- Collecting characters into integers
- Computing their collective numerical value
- Numbers can be treated as single units during parsing and translation

$31 + 28 + 59 \rightarrow \langle \text{num, 31} \rangle \langle + \rangle \langle \text{num, 28} \rangle \langle + \rangle \langle \text{num, 59} \rangle$
if ( peek holds a digit ) {
    v = 0;
    do {
        v = v * 10 + integer value of digit peek;
        peek = next input character;
    } while ( peek holds a digit );
    return token ⟨num, v⟩;
}
Recognizing Keywords and Identifiers

Grammars treat identifiers as terminals

Example: \( \text{count} = \text{count} + \text{increment}; \)

treated as

\( \text{id} = \text{id} + \text{id} \)

\( \langle \text{id}, \text{"count"} \rangle \langle = \rangle \langle \text{id}, \text{"count"} \rangle \langle + \rangle \langle \text{id}, \text{"increment"} \rangle \langle ; \rangle \)
Recognizing Keywords and Identifiers

• A mechanism is needed to decide when a lexeme is an identifier or a keyword
• Life is much easier if keywords are reserved
• The best way is to store them in a table
  – String table
  – An entry is a string and a token
• Initialize the table with keywords
if ( peek holds a letter ) {
    collect letters or digits into a buffer \( b \);
    \( s \) = string formed from the characters in \( b \);
    \( w \) = token returned by \( \text{words.get}(s) \);
    if ( \( w \) is not \text{null} ) return \( w \);
    else {
        Enter the key-value pair \( (s, (\text{id}, s)) \) into \( \text{words} \)
        return token \( (\text{id}, s) \);
    }
}
Symbol Tables

• Data structures used by compilers to hold information about source program constructs
• Scope is an important issue here
  – Most-closely nested rule
• Symbol table per scope
{ int x_1; int y_1;
  { int w_2; bool y_2; int z_2;
     \ldots w_2 \ldots; \ldots x_1 \ldots; \ldots y_2 \ldots; \ldots z_2 \ldots;
  }
  \ldots w_0 \ldots; \ldots x_1 \ldots; \ldots y_1 \ldots;
}

\begin{figure}
\centering
\begin{tikzpicture}
  \node (B0) at (0,0) {
    \begin{tabular}{c|c|c}
      \hline
      w & & \\
      \cdashline{1-3}
      \cdashline{1-3}
      \cdashline{1-3}
      \cdashline{1-3}
      \ \ \ \ \ ...
    \end{tabular}
  }
  \node (B1) at (1,-1) {
    \begin{tabular}{c|c|c}
      \hline
      x & int & \\
      \hline
      y & int & \\
    \end{tabular}
  }
  \node (B2) at (1,-2) {
    \begin{tabular}{c|c|c|c|c}
      \hline
      w & int & & & \\
      \hline
      y & bool & & & \\
      \hline
      z & int & & & \\
    \end{tabular}
  }
  \draw (B0) -- (B1);
  \draw (B1) -- (B2);
\end{tikzpicture}
\end{figure}
{ int x_1; int y_1;
  { int w_2; bool y_2; int z_2; ... w_2 ...; ... x_1 ...; ... y_2 ...; ... z_2 ...;
    ... w_0 ...; ... x_1 ...; ... y_1 ...;
  }
}
How Are Symbol Tables Accessed?

• Using semantic action
• A semantic action can put information in symbol table
• A semantic action can get information from symbol table
program $\rightarrow$ block

block $\rightarrow$ '{' $\{top = null;\}$

$\{\text{ saved } = top;\}$
$\text{ top } = \text{ new Env}(top);$
$\text{ print("{ "}); }$

$\{\text{ top } = \text{ saved; }\}$
$\text{ print("} "$); }$

decls $\rightarrow$ decls decl
$\mid\epsilon$

decl $\rightarrow$ type id ;
\{$s = \text{ new Symbol; }$
\$\text{ s.type } = \text{ type.lexeme} \$
\$\text{ top.put(id.lexeme, s); }\$

stmts $\rightarrow$ stmts stmt
$\mid\epsilon$

stmt $\rightarrow$ block
$\mid$ factor ;
\{$\text{ print(" ; "); }\$

factor $\rightarrow$ id
\{$s = \text{ top.get(id.lexeme);} \$
$\text{ print(id.lexeme);} \$
$\text{ print(" : "); }$
$\text{ print(s.type);} \$
Intermediate Code Generation

• Two kinds
  – Trees
    • parse tree
    • abstract syntax tree

• Linear representations
  – three-address code
  – Needed if we want to do optimizations
Static Checking

• Static because done at compile time
• Syntactic checking
  – more than grammar
  – example: break must be in a loop, identifier must be declared, ...
• Type checking
  – Ensures that an operator or function is applied to the right number and type of operands
More On type Checking

- **L-values and R-values**
  - L-values are locations
  - R-values are “values”

- **Matching actual with expected values**
  - *Coercion*: type of an operand is automatically converted to the type expected by the operator
  - *Overloading*: symbol has different meaning depending on context
We Are Done With Chapter 2!

• Read 2.4 -> 2.8
  – skim: 2.5.4, 2.5.5, 2.6.5, 2.8.4
  – Ready carefully the rest

• You can skim over the implementations in java in some of the sections, they are useful

• Why the final exam is not tomorrow?