CSCI-GA.2130-001
Compiler Construction
Lecture 3:
Syntax-Directed Translator (Cont’d)

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
A Quick Summary

Attributes

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

```
expr → expr + term  |  term → 0
|  expr - term  |  1
|  term         |  ...  
|               |  9
```

```
expr.t = 95-2+

expr.t = 95-  +   term.t = 2

expr.t = 9

- term.t = 5  2

term.t = 9

5

term.t = 9

9
```
A Quick Summary

Translation schemes

- Instead of attributes add program fragment to production rules
- They are called semantic actions

\[
\begin{align*}
expr & \rightarrow expr + term \\
       & \rightarrow expr - term \\
       & \rightarrow term \\
\end{align*}
\]

\[
\begin{align*}
term & \rightarrow 0 \\
       & \rightarrow 1 \\
       & \rightarrow \ldots \\
       & \rightarrow 9 \\
\end{align*}
\]

\[
\begin{align*}
expr & \rightarrow expr_1 + term \{\text{print(‘+’)}\} \\
expr & \rightarrow expr_1 - term \{\text{print(‘-’)}\} \\
expr & \rightarrow term \\
term & \rightarrow 0 \{\text{print(‘0’)}\} \\
term & \rightarrow 1 \{\text{print(‘1’)}\} \\
       & \rightarrow \ldots \\
term & \rightarrow 9 \{\text{print(‘9’)}\}
\end{align*}
\]
Example

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.
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.
Predictive Parsing

Terminal is matched
Lookahead advances.

\[
\text{stmt} \rightarrow \begin{cases} 
\text{expr;} & \text{if ( expr ) stmt} \\
\text{for ( optexpr ; optexpr ; optexpr )} & \text{other} \\
\epsilon & \text{expr} 
\end{cases}
\]

\[
\text{void stmt() \{} \\
\quad \text{switch ( lookahead ) \{} \\
\quad \quad \text{case expr:} \\
\quad \quad \quad \text{match(expr); match(';'); break;}
\quad \quad \text{case if:}
\quad \quad \quad \text{match(if); match('('); match(expr); match(')'); stmt(); break;}
\quad \quad \text{case for:}
\quad \quad \quad \text{match(for); match('(');
\quad \quad \quad \quad \text{optexpr(); match('); optexpr(); match('); optexpr();
\quad \quad \quad \quad \quad \text{match(')'); stmt(); break;}
\quad \quad \text{case other;}
\quad \quad \quad \text{match(other); break;}
\quad \text{default:}
\quad \quad \quad \text{report("syntax error");}
\quad \}\}
\}\}
\]

\[
\text{void optexpr() \{} \\
\quad \text{if ( lookahead == expr ) match(expr);}
\}\}
\]

\[
\text{void match(terminal t) \{} \\
\quad \text{if ( lookahead == t ) lookahead = nextTerminal;}
\quad \text{else report("syntax error");}
\}\}
\]
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

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

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

<table>
<thead>
<tr>
<th>expr</th>
<th>→</th>
<th>expr + term</th>
<th>{ print('+') }</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>expr - term</td>
<td>{ print('-') }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>term</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>term</th>
<th>→</th>
<th>0</th>
<th>{ print('0') }</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>{ print('1') }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>{ print('9') }</td>
</tr>
</tbody>
</table>

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

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

\[
A \rightarrow A\alpha \mid A\beta \mid \gamma \\
A \rightarrow \gamma R \\
R \rightarrow \alpha R \mid \beta R \mid \varepsilon
\]
Let’s Build A Translator: Arithmetic Expressions to Postfix

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

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

\[
\begin{align*}
expr & \rightarrow \text{term rest} \\
\text{rest} & \rightarrow \text{+ term} \setminus \{ \text{print('+'}) \} \text{ rest} \\
& \mid \text{− term} \setminus \{ \text{print('-')} \} \text{ rest} \\
& \mid \epsilon \\
\text{term} & \rightarrow 0 \setminus \{ \text{print('0')} \} \\
& \mid 1 \setminus \{ \text{print('1')} \} \\
& \mid \ldots \\
& \mid 9 \setminus \{ \text{print('9')} \}
\end{align*}
\]

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

\[
\begin{align*}
A & \rightarrow \gamma R \\
R & \rightarrow \alpha R \mid \beta R \mid \epsilon
\end{align*}
\]

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 distinction
\[
\begin{align*}
expr & \rightarrow \ expr + \ term \quad \{ \text{print(')+'}) \} \\
& \mid expr - term \quad \{ \text{print(')-'}) \} \\
& \mid \ \ \ \ \ \ \ \ \ \ \ \ term \\
\text{term} & \rightarrow \ term \ast \ factor \quad \{ \text{print(')*'}) \} \\
& \mid term / factor \quad \{ \text{print(')/'}) \} \\
& \mid \ \ \ \ \ \ \ \ \ \ \ \ factor \\
\text{factor} & \rightarrow \ ( \ expr \ ) \\
& \mid \ \text{num} \quad \{ \text{print(num.value)} \} \\
& \mid \ \text{id} \quad \{ \text{print(id.lexeme)} \}
\end{align*}
\]
Thanks to the lexical analyzer, the parser can deal with any number.

Thanks to the lexical analyzer, the parser can deal with identifier.
Issues in Lexical Analysis

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

Makes parser’s life much easier

```
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
  – It reads ahead only when it must
• 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: count = count + increment;
treated as

$id = id + id$

\[(id, "count") \rightarrow (id, "count") \rightarrow (id, "increment") \rightarrow (;)\]
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 words.get(s);
    if ( w is not null ) return w;
    else {
        Enter the key-value pair (s, (id, s)) into words
        return token (id, s);
    }
}
Symbol Tables

• Data structure used by compilers to hold information about source program constructs

• Scope is an important issue here
  – Symbol table per scope
\{ 
  \textbf{int} \ x_1; \ \textbf{int} \ y_1; \\
  \{  \textbf{int} \ w_2; \ \textbf{bool} \ y_2; \ \textbf{int} \ z_2; \\
       \cdots w_2 \cdots; \ \cdots x_1 \cdots; \ \cdots y_2 \cdots; \ \cdots z_2 \cdots; \\
  \} \\
  \cdots w_0 \cdots; \ \cdots x_1 \cdots; \ \cdots y_1 \cdots; \\
\} \\

B_0:\begin{array}{c|c|c|c}
  & w & \\
\hline
\cdots & & \\
\end{array} \\

B_1:\begin{array}{c|c|c|c}
  x & \textbf{int} & \\
\hline
  y & \textbf{int} & \\
\end{array} \\

B_2:\begin{array}{c|c|c|c}
  w & \textbf{int} & \\
\hline
  y & \textbf{bool} & \\
  z & \textbf{int} & \\
\end{array}
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 \ \ ; \ 
}

B_0: \begin{array}{c}
w \\
\ldots
\end{array}

B_1: \begin{array}{c}
x \ \text{int} \\
y \ \text{int}
\end{array}

B_2: \begin{array}{c}
w \ \text{int} \\
y \ \text{bool} \\
z \ \text{int}
\end{array}
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 → block
           { top = null; }

block → '{
         { saved = top;
           top = new Env(top);
           print("{ ""); }
        decls stmts '}'
        { top = saved;
          print("}"); }

decls → decls decl
       | ε

decl → type id ;
      { s = new Symbol;
        s.type = type.lexeme
        top.put(id.lexeme, s); }

stmts → stmts stmt
       | ε

stmt → block
      | factor ;
      { print("; "); }

factor → id
      { s = top.get(id.lexeme);
        print(id.lexeme);
        print(":"); }
      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, 2.8.2, and 2.8.4
  – Read 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?