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
| expr - term
| term

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr → expr₁ + term</td>
<td>expr.t = expr₁.t</td>
</tr>
<tr>
<td>expr → expr₁ - term</td>
<td>expr.t = expr₁.t</td>
</tr>
<tr>
<td>expr → term</td>
<td>expr.t = term.t</td>
</tr>
<tr>
<td>term → 0</td>
<td>term.t = '0'</td>
</tr>
<tr>
<td>term → 1</td>
<td>term.t = '1'</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>term → 9</td>
<td>term.t = '9'</td>
</tr>
</tbody>
</table>

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

expr.t = 9 - term.t = 5 + 2

term.t = 9 - 5

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 \\
& \quad | expr - term \\
& \quad | term \\

digit & \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\end{align*}
\]

\[
\begin{align*}
expr & \rightarrow expr_1 + term \quad \text{\{print('+'\}\}} \\
expr & \rightarrow expr_1 - term \quad \text{\{print('-'\}\}} \\
expr & \rightarrow term \\
term & \rightarrow 0 \quad \text{\{print('0'\}\}} \\
term & \rightarrow 1 \quad \text{\{print('1'\}\}} \\
& \quad \ldots \\
term & \rightarrow 9 \quad \text{\{print('9'\}\}}
\end{align*}
\]
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.
Predictive Parsing

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

\[ optexpr \rightarrow \epsilon \\
| \quad expr \]

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

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

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

Each nonterminal becomes a procedure.
Predictive Parsing

```
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");
}
```

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

expr -> expr + term

This can loop forever. Can you see why?

We can eliminate $A \rightarrow A\alpha | \beta$ as follows:

$$
A \rightarrow \beta R \\
R \rightarrow \alpha R | \epsilon
$$
Let's Build A Translator: Arithmetic Expressions to Postfix

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

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

Do you see any problems with this production?
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’) }

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

\[
expr \rightarrow expr + \text{term} \quad \{ \text{print}(\text{'+'}) \} \\
| \quad expr - \text{term} \quad \{ \text{print}(\text{'-'} \} \\
| \quad \text{term} \\
\]

\[
term \rightarrow 0 \quad \{ \text{print}(\text{'0'} \} \\
| \quad 1 \quad \{ \text{print}(\text{'1'} \} \\
| \quad \ldots \\
| \quad 9 \quad \{ \text{print}(\text{'9'} \} \\
\]

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

\[
A = expr \\
\alpha = + \text{term} \{ \text{print}(\text{'+'}) \} \\
\beta = - \text{term} \{ \text{print}(\text{'-'} \} \\
\gamma = \text{term}
\]
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
\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{term} \quad \{ \text{print}(\'+\') \} \\
& \ | \quad \text{expr} - \text{term} \quad \{ \text{print}('-' ) \} \\
& \ | \quad \text{term} \\
\text{term} & \rightarrow \text{term} \ast \text{factor} \quad \{ \text{print}(\'*\') \} \\
& \ | \quad \text{term} \div \text{factor} \quad \{ \text{print}('/' ) \} \\
& \ | \quad \text{factor} \\
\text{factor} & \rightarrow ( \text{expr} ) \\
& \ | \quad \text{num} \quad \{ \text{print}(\text{num}.\text{value} ) \} \\
& \ | \quad \text{id} \quad \{ \text{print}(\text{id}.\text{lexeme} ) \}
\end{align*}
\]
Thanks to the lexical analyzer, the parser can deal with identifier and 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

```
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  →  \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 {
    \( v = v \times 10 + \text{integer value of digit } \text{peek}; \)
    \( \text{peek} = \text{next input character}; \)
  } while ( \( \text{peek holds a digit} \) );
  return token \langle \text{num}, v \rangle;
}
Recognizing Keywords and Identifiers

Grammars treat identifiers as terminals

Example: `count = count + increment;`
treated as

`id = id + id`

\[
(id, "count") \rightarrow (id, "count") (+) (id, "increment") (;)
\]
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 structures used by compilers to hold information about source program constructs

• Scope is an important issue here
  – Symbol table per scope
{    int x₁; int y₁;
    { int w₂; bool y₂; int z₂;
      ⋮ w₂ ⋮; ⋮ x₁ ⋮; ⋮ y₂ ⋮; ⋮ z₂ ⋮;
    }
    ⋮ w₀ ⋮; ⋮ x₁ ⋮; ⋮ y₁ ⋮;
int $x_1$; int $y_1$;

\{
  int $w_2$; bool $y_2$; int $z_2$;
  ⋮ $w_2$ ⋮; ⋮ $x_1$ ⋮; ⋮ $y_2$ ⋮; ⋮ $z_2$ ⋮;
  ⋮ $w_0$ ⋮; ⋮ $x_1$ ⋮; ⋮ $y_1$ ⋮;
\}

$B_0$:

\[
\begin{array}{c}
  w \\
  ⋮ \\
\end{array}
\]

$B_1$:

\[
\begin{array}{c|c}
  x & \text{int} \\
  y & \text{int} \\
\end{array}
\]

$B_2$:

\[
\begin{array}{c|c}
  w & \text{int} \\
  y & \text{bool} \\
  z & \text{int} \\
\end{array}
\]
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 

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?