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
From Last Time

Translation schemes

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

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

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

\[
\begin{align*}
\text{stmt} & \rightarrow \text{expr} ; \\
& \mid \text{if ( expr ) stmt} \\
& \mid \text{for ( optexpr ; optexpr ; optexpr )} \\
& \mid \text{other}
\end{align*}
\]

\[
\begin{align*}
\text{optexpr} & \rightarrow \epsilon \\
& \mid \text{expr}
\end{align*}
\]

```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

Each nonterminal becomes a procedure.
Predictive Parsing

```
stmt  →  expr ;
    |    if ( expr ) stmt
    |    for ( optexpr ; optexpr ; optexpr )
    |    other
optexpr  →  ε
        |  expr

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:

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

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

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

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  →  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

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

term → term * factor \{ print('*') \}  
| term / factor \{ print('/') \}  
| factor  

factor → ( expr )  
| num \{ print(num.value) \}  
| id \{ print(id.lexeme) \}
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

```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
  – 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

$31 + 28 + 59 \quad \langle \text{num,31} \rangle \langle + \rangle \langle \text{num,28} \rangle \langle + \rangle \langle \text{num,59} \rangle$
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`

\[
\langle \text{id}, "count" \rangle = \langle \text{id}, "count" \rangle + \langle \text{id}, "increment" \rangle ;
\]

lexeme
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 \texttt{words.get}(s);
    if ( \( w \) is not \texttt{null} ) return \( w \);
    else {
        Enter the key-value pair \((s, \langle \texttt{id}, s \rangle)\) into \texttt{words}
        return token \( \langle \texttt{id}, s \rangle \);
    }
}
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
\{
    \text{int } x_1; \text{ int } y_1;
    \{
        \text{int } w_2; \text{ bool } y_2; \text{ 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{align*}
    B_0: & \quad \begin{array}{|c|}
    \hline
    w \\
    \hline
    \ldots \\
    \hline
    \end{array} \\
    B_1: & \quad \begin{array}{|c|}
    \hline
    x \text{ int} \\
    \hline
    y \text{ int} \\
    \hline
    \end{array} \\
    B_2: & \quad \begin{array}{|c|}
    \hline
    w \text{ int} \\
    \hline
    y \text{ bool} \\
    \hline
    z \text{ int} \\
    \hline
    \end{array}
\end{align*}
\{ 
\{ 
    \textbf{int} \ x_1; \ \textbf{int} \ y_1; \\
    \textbf{int} \ w_2; \ \textbf{bool} \ y_2; \ \textbf{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{array}{|c|c|}
\hline
w & \ldots \\
\hline
\ldots & \\
\hline
\end{array} 

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

\begin{array}{|c|c|c|}
\hline
w & \textbf{int} & \\
\hline
y & \textbf{bool} & \\
\hline
z & \textbf{int} & \\
\hline
\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 → '{' {
  top = null;
  
  saved = top;
  top = new Env(top);
  print("{ ");
}
decs stmts '}' {
  top = saved;
  print("} ");
}

decs → decs 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?