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

\[
expr \rightarrow expr + term \\
| expr - term \\
| term
\]
From Last Time

Translation schemes

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

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

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

\[
\text{optexpr} \rightarrow \epsilon \\
| \text{expr}
\]

```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");
}
Predictive Parsing

Each nonterminal becomes a procedure.

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
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

\[
\begin{align*}
expr & \rightarrow expr + term \quad \{ \text{print('}\mathbf{+}') \} \\
& \quad \mid expr - term \quad \{ \text{print('}-') \} \\
& \quad \mid term

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*}
\]
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') }

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

\[
\begin{align*}
expr & \rightarrow \text{expr } + \text{ term} \quad \{ \text{print(‘+’)} \} \\
& \quad | \quad \text{expr } - \text{ term} \quad \{ \text{print(‘-’)} \} \\
& \quad | \quad \text{term} \\
term & \rightarrow \text{0} \quad \{ \text{print(‘0’)} \} \\
& \quad | \quad \text{1} \quad \{ \text{print(‘1’)} \} \\
& \quad \quad \quad \ldots \\
& \quad | \quad \text{9} \quad \{ \text{print(‘9’)} \} \\
A & \rightarrow A\alpha \mid A\beta \mid \gamma \\
A & \rightarrow \gamma R \\
R & \rightarrow \alpha R \mid \beta R \mid \epsilon
\end{align*}
\]
Let’s Build A Translator: Arithmetic Expressions to Postfix

\[
\begin{align*}
expr & \to \; expr + term \quad \{ \text{print}('+') \} \\
& \mid \; expr - term \quad \{ \text{print}('-') \} \\
& \mid \; term \\

term & \to \; 0 \quad \{ \text{print}('0') \} \\
& \mid \; 1 \quad \{ \text{print}('1') \} \\
& \mid \; ... \\
& \mid \; 9 \quad \{ \text{print}('9') \}
\end{align*}
\]

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

\[
\begin{align*}
A & = \text{expr} \\
\alpha & = \text{+ term} \; \{ \text{print}('+') \} \\
\beta & = \text{- term} \; \{ \text{print}('-') \} \\
\gamma & = \text{term}
\end{align*}
\]
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
\[ \text{expr} \rightarrow \text{expr} + \text{term} \quad \{ \text{print('+')} \} \\
| \quad \text{expr} - \text{term} \quad \{ \text{print('-')} \} \\
| \quad \text{term} \\
\]

\[ \text{term} \rightarrow \text{term} \ast \text{factor} \quad \{ \text{print('*')} \} \\
| \quad \text{term} / \text{factor} \quad \{ \text{print('/')} \} \\
| \quad \text{factor} \\
\]

\[ \text{factor} \rightarrow (\text{expr}) \\
| \quad \text{num} \quad \{ \text{print(num.value)} \} \\
| \quad \text{id} \quad \{ \text{print(id.lexeme)} \} \]
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

```plaintext
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 ) {
    \[ v = 0; \]
    \textbf{do} {
        \[ v = v \times 10 + \ \text{integer value of digit \textit{peek};} \]
        \textit{peek} = next input character;
    } \textbf{while} ( \textit{peek} holds a digit );
    return token \langle \textbf{num}, v \rangle ;
}
Recognizing Keywords and Identifiers

Grammars treat identifiers as terminals

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

treated as

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

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

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 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
\[
\{ \quad \text{int } x_1; \quad \text{int } y_1; \\
\{ \quad \text{int } w_2; \quad \text{bool } y_2; \quad \text{int } z_2; \\
\quad \cdots w_2 \cdots; \quad \cdots x_1 \cdots; \quad \cdots y_2 \cdots; \quad \cdots z_2 \cdots; \\
\} \\
\quad \cdots w_0 \cdots; \quad \cdots x_1 \cdots; \quad \cdots y_1 \cdots; \\
\}\]

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

\[
B_1 : \quad \begin{array}{c|c|c|c}
x & \text{int} & & \\
\hline
y & \text{int} & & \\
\end{array}
\]

\[
B_2 : \quad \begin{array}{c|c|c|c}
w & \text{int} & & \\
\hline
y & \text{bool} & & \\
z & \text{int} & & \\
\end{array}
\]
\begin{verbatim}
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 ...;
\end{verbatim}
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 → '{' decls stmts '}'

  decls → decls decl | ε

  decl → type id ;

  stmts → stmts stmt | ε

  stmt → block | factor ;

  factor → id
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?