CSCI-GA.2130-001
Compiler Construction
Lecture 2:
Syntax-Directed Translator

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
What Will We Do?

• Build a very simple compiler
• Only the front end
  – Code generation
• Easy and limited source language
• Will touch upon everything quickly
• Chapters 3-8 give more details
ANALYSIS PHASE

- Break your program into pieces
- Produce an internal presentation of it
Allows a translator to handle multicharacter constructs
Abstract Syntax Tree

source program → Lexical Analyzer → tokens → Parser → syntax tree → Intermediate Code Generator → three-address code

Symbol Table

do-while

body

assign

[ ] v

i + a i

i 1
• A data structure
• Hold information about source code constructs
• Information collected incrementally at analysis phase
• Used by synthesis phase
1: \( i = i + 1 \)
2: \( t1 = a[i] \)
3: if \( t1 < v \) goto 1
4: \( j = j - 1 \)
5: \( t2 = a[j] \)
6: if \( t2 > v \) goto 4
7: ifFalse \( i >= j \) goto 9
8: goto 14
9: \( x = a[i] \)
10: \( t3 = a[j] \)
11: \( a[i] = t3 \)
12: \( a[j] = x \)
13: goto 1
14:
How Do We Define Language Syntax?

• Using a special notation
• Context-free grammar
• Set of rules

Example:

\textbf{If} ( \text{expression} ) \text{ statement else statement}

Corresponds to a rule:

\texttt{stmt} -> \texttt{if} (\texttt{expr}) \texttt{stmt else stmt}
Production Rules

stmt -> if (expr) stmt else stmt

- head or left hand side (LHS)
- body or right hand side

may be read as:

can have the form
Production Rules

`stmt -> if (expr) stmt else stmt`

Nonterminals

They need more rules to define them.
Production Rules

stmt -> if (expr) stmt else stmt

Terminals
No more rules needed for them
Components of Context-Free Grammar

- Set of **terminal** symbols
- Set of **nonterminals**
- Set of **productions**
  - The head is nonterminal
  - The body is a sequence of terminals and/or nonterminals
- Designation of one nonterminal as **starting symbol**
Example

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

What are the terminals here?

What are the nonterminals?

What does this grammar generate?
Some Definitions

• **String of terminals**: sequence of zero or more terminals

• **Derivation**:
  – given the grammar (i.e. productions)
  – begin with the start symbol
  – repeatedly replacing nonterminal by the body
  – We obtain the language defined by the grammar (i.e. group of terminal strings)

• **Parsing**:
  – Given a string of terminals
  – Figure out how to derive it from the start symbol of the grammar
Example

\[
\text{list} \rightarrow \text{list} + \text{digit} \mid \text{list} - \text{digit} \mid \text{digit}
\]
\[
\text{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

How to derive: 9-5+7 from the above rules?
Parse Tree

- Pictorially shows how the start symbol of a grammar derives a given string

A -> XYX

- Root is labeled by the start symbol
- Interior nodes are nonterminals
- Each leave is a terminal or ε

The process of finding a parse tree for a given string of terminals is called parsing.
Example

Deriving $9 - 5 + 2$ from

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

Can we derive $9 - 5 + 2$ from

\[
\text{string} \rightarrow \text{string} + \text{string} \mid \text{string} - \text{string} \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]
Ambiguity

- A grammar is ambiguous if it has more than one parse tree generating the same string of terminals.

Two parse trees for 9−5+2

```
string → string + string | string − string | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```
Example

Is the following grammar ambiguous?

\[ S \rightarrow +SS \mid -SS \mid a \]
Example

Is the following grammar ambiguous?

\[ S \rightarrow S(S)S \mid \varepsilon \]
Example

Is the following grammar ambiguous?

\[ S \rightarrow a \mid S + S \mid SS \mid S^* \mid (S) \]
Associativity of Operators

How will you evaluate this? 9-5-2

Will ‘5’ go with the ‘-’ on the left or the one on the right?

If it goes with the one on the left: (9-5)-2 we say that the operator ‘-’ is left-associative

If it goes with the one on the right: 9-(5-2) we say that the operator ‘-’ is right-associative
Associativity of Operators

How to express associativity in production rules?

term -> term - digit|digit

digit -> 0|1|2|3|4|5|6|7|8|9

Left-associative
(9-5)-2

term -> digit-term|digit
digit -> 0|1|2|3|4|5|6|7|8|9

Right-associative
9-(5-2)
Precendence of Operators

• Associativity applies to occurrence of the same operator
• What if operators are different?
• How will you evaluate: 9-5*2
• We say ‘*’ has higher precedence than ‘-‘ if it takes its operands before ‘-‘
Precedence of Operators

How to present this in productions?

\[
\begin{aligned}
expr & \rightarrow expr + term \\
& \quad | expr - term \\
& \quad | term \\
\end{aligned}
\]

\[
\begin{aligned}
term & \rightarrow term * factor \\
& \quad | term / factor \\
& \quad | factor \\
\end{aligned}
\]

\[
\begin{aligned}
factor & \rightarrow digit \mid ( expr )
\end{aligned}
\]

The above example shows both precedence and associativity

* / have higher precedence then + -

All of them are left associative
Example

Construct unambiguous context-free grammar for left-associate list of identifiers separated by commas.
Syntax-Directed Translation

• We have built a parse-tree, now what?
• How will this tree and production rules help in translation?
• This means we have to associate *something* with each production and with each tree node
Syntax-Directed Translation

• **Attributes**
  – Each symbol (terminal or nonterminal) has an attribute
  – Semantic rules for calculating attributes of a node from its children

• **Translation scheme is a notation for attaching program fragments to productions**
\[ \begin{align*}
  \text{expr} & \rightarrow \text{expr} + \text{term} \\
  & \mid \text{expr} - \text{term} \\
  & \mid \text{term} \\
  \text{term} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{align*} \]

\[ \text{expr and term each has an attribute: expr.t and term.t} \]

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{expr} \rightarrow \text{expr}_1 + \text{term} )</td>
<td>( \text{expr.t} = \text{expr}_1.t | \text{term.t} | '+' )</td>
</tr>
<tr>
<td>( \text{expr} \rightarrow \text{expr}_1 - \text{term} )</td>
<td>( \text{expr.t} = \text{expr}_1.t | \text{term.t} | '-' )</td>
</tr>
<tr>
<td>( \text{expr} \rightarrow \text{term} )</td>
<td>( \text{expr.t} = \text{term.t} )</td>
</tr>
<tr>
<td>( \text{term} \rightarrow 0 )</td>
<td>( \text{term.t} = '0' )</td>
</tr>
<tr>
<td>( \text{term} \rightarrow 1 )</td>
<td>( \text{term.t} = '1' )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \text{term} \rightarrow 9 )</td>
<td>( \text{term.t} = '9' )</td>
</tr>
</tbody>
</table>

\[ \| \| \text{ means concatenate} \]
Attribute values at nodes for 9-5+2

- Build the tree
- Start from leaves
- Using semantic rules till you reach root

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr → expr1 + term</code></td>
<td>`expr.t = expr1.t</td>
</tr>
<tr>
<td><code>expr → expr1 - term</code></td>
<td>`expr.t = expr1.t</td>
</tr>
<tr>
<td><code>expr → term</code></td>
<td><code>expr.t = term.t</code></td>
</tr>
<tr>
<td><code>term → 0</code></td>
<td><code>term.t = '0'</code></td>
</tr>
<tr>
<td><code>term → 1</code></td>
<td><code>term.t = '1'</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>term → 9</code></td>
<td><code>term.t = '9'</code></td>
</tr>
</tbody>
</table>

Figure 2.10: Syntax-directed definition for infix to postfix translation

Attributes can be evaluated during a single bottom-up traversal of a parse tree.
Another Way: Translation Schemes

• Another notation
• Attaching program fragments to productions
• These program fragments are called semantic actions

example: \( rest \rightarrow + \ term \ \{ \text{print('+' )}\} \ rest_1 \)
9 - 5 + 2
9-5+2

With semantic actions

With attributes
Concerning Tree Traversal

Depth first
• Preorder
• Postorder
Back to Parsing!

- We have a set of productions
- We have a string of terminals
- We need to form the parse-tree that will generate that string
Given this set of productions:  

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

and this string:  

```
for( ; expr ; expr ) other
```

How can we generate this?
\[
\begin{align*}
stmt & \rightarrow \text{expr ;} \\
& \mid \text{if ( expr ) stmt} \\
& \mid \text{for ( optexpr ; optexpr ; optexpr ) stmt} \\
& \mid \text{other}
\end{align*}
\]

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

**PARSE TREE**

(a)  

**INPUT**  

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

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

**PARSE TREE**

```
\( \text{for ( optexpr ; optexpr ; optexpr ) stmt} \)
```

**INPUT**

```
\( \text{for ( ; expr ; expr ) other} \)
```
Note: Sometimes choosing the right production may involve trial and error, and backtracking
Parsing With No-Backtracking

- Top-down method
- Based on recursive procedures
- Part of a parsing category called: Recursive-descent parsing
  - The lookahead symbol unambiguously determines the flow-of control
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");
}
Designing Predictive Parser

• By examining the lookahead symbol we choose a production
• There must not be any conflict between two bodies with same head otherwise we cannot use predictive-parsing
• The procedure mimics the body of the chosen production
  – nonterminal is a procedure call
  – terminal is matched and lookahead advances
Example

\[ \text{expr} \rightarrow \text{expr} + \text{term} \mid \text{term} \]

\[ \begin{align*}
\text{term} \\
\text{term} + \text{term} \\
\text{term} + \text{term} + \text{term} \\
\ldots
\end{align*} \]

\[ \begin{align*}
\text{expr} & \rightarrow \text{term} \text{ factor} \\
\text{factor} & \rightarrow + \text{ term} \text{ factor} \mid \epsilon
\end{align*} \]
Enough for Today

• Next time we will continue our trip for building simple translator
• This lecture covered 2.1 -> 2.4