NEW YORK UNIVERSITY

CSCI-GA.2130-001

Compiler Construction

Lecture 3: Syntax-Directed Translator (Cont'd)

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

A Quick Summary

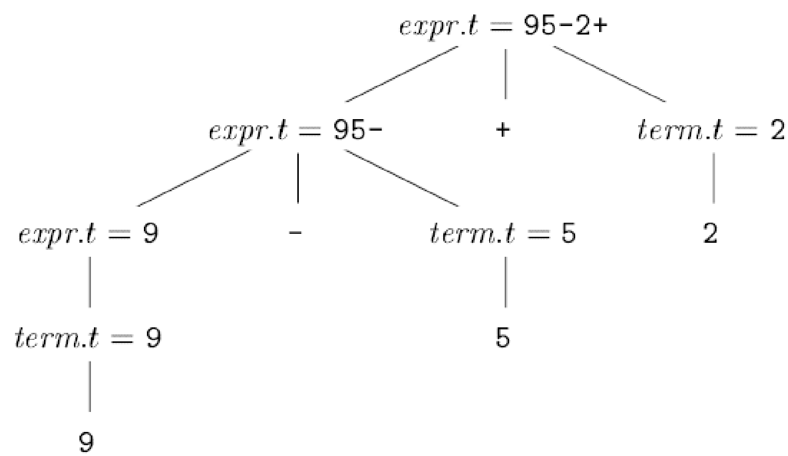
Attributes

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

$$
\begin{array}{lcl}
expr & \rightarrow & expr + term \\
 & | & expr - term \\
 & | & term
\end{array}
$$

$$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$



A Quick Summary

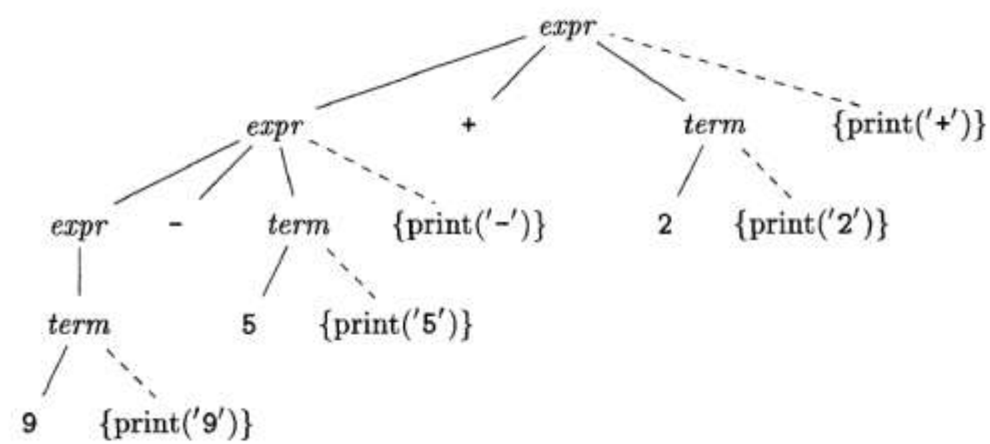
Translation schemes

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

$$
\begin{array}{lcl}
expr & \rightarrow & expr + term \\
 & | & expr - term \\
 & | & term
\end{array}
$$

$$digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$
\begin{array}{lcll}
expr & \rightarrow & expr_1 + term & \{\text{print}('+')\} \\
expr & \rightarrow & expr_1 - term & \{\text{print}('-')\} \\
expr & \rightarrow & term & \\
term & \rightarrow & 0 & \{\text{print}('0')\} \\
term & \rightarrow & 1 & \{\text{print}('1')\} \\
 & & \ldots & \\
term & \rightarrow & 9 & \{\text{print}('9')\}
\end{array}
$$



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 → **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");
}
```

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");
}
```

Each nonterminal becomes a procedure.

Predictive Parsing

$$
\begin{aligned}
stmt &\rightarrow \textbf{expr} \; ; \\
&\mid \; \textbf{if} \; (\; \textbf{expr} \;) \; stmt \\
&\mid \; \textbf{for} \; (\; optexpr \; ; \; optexpr \; ; \; optexpr \;) \\
&\mid \; \textbf{other} \\
optexpr &\rightarrow \epsilon \\
&\mid \; \textbf{expr}
\end{aligned}
$$

```
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 $A \rightarrow A\alpha \mid \beta$ As follows:

$$
\begin{aligned}
A &\rightarrow \beta R \\
R &\rightarrow \alpha R \mid \epsilon
\end{aligned}
$$

Let's Build A Translator: Arithmetic Expressions to Postfix

$$
\begin{array}{lcll}
expr & \rightarrow & expr + term & \{\ \mathrm{print}('+')\ \} \\
 & | & expr - term & \{\ \mathrm{print}('-')\ \} \\
 & | & term & \\
 & & & \\
term & \rightarrow & 0 & \{\ \mathrm{print}('0')\ \} \\
 & | & 1 & \{\ \mathrm{print}('1')\ \} \\
 & & \dots & \\
 & | & 9 & \{\ \mathrm{print}('9')\ \}
\end{array}
$$

Do you see any problems with this production?

Let's Build A Translator: Arithmetic Expressions to Postfix

$$
\begin{array}{lcll}
expr & \rightarrow & expr + term & \{\ \text{print}('+')\ \} \\
 & | & expr - term & \{\ \text{print}('-')\ \} \\
 & | & term & \\
 & & & \\
term & \rightarrow & 0 & \{\ \text{print}('0')\ \} \\
 & | & 1 & \{\ \text{print}('1')\ \} \\
 & & \ldots & \\
 & | & 9 & \{\ \text{print}('9')\ \}
\end{array}
$$

$$
A \rightarrow A\alpha \mid \beta
$$

$$
\begin{array}{lcl}
A & \rightarrow & \beta R \\
R & \rightarrow & \alpha R \mid \epsilon
\end{array}
$$

Can we apply the above rule here?

Let's Build A Translator: Arithmetic Expressions to Postfix

$$
\begin{array}{lcll}
expr & \rightarrow & expr + term & \{\ \text{print}('+')\ \} \\
 & | & expr - term & \{\ \text{print}('-')\ \} \\
 & | & term & \\
 & & & \\
term & \rightarrow & 0 & \{\ \text{print}('0')\ \} \\
 & | & 1 & \{\ \text{print}('1')\ \} \\
 & & \ldots & \\
 & | & 9 & \{\ \text{print}('9')\ \}
\end{array}
$$

$$A \rightarrow A\alpha \mid A\beta \mid \gamma$$

$$
\begin{array}{lcl}
A & \rightarrow & \gamma R \\
R & \rightarrow & \alpha R \mid \beta R \mid \epsilon
\end{array}
$$

Let's Build A Translator: Arithmetic Expressions to Postfix

$$
\begin{array}{lcll}
expr & \rightarrow & expr + term & \{\ \mathrm{print}('+')\ \} \\
 & | & expr - term & \{\ \mathrm{print}('-')\ \} \\
 & | & term & \\
 & & & \\
term & \rightarrow & 0 & \{\ \mathrm{print}('0')\ \} \\
 & | & 1 & \{\ \mathrm{print}('1')\ \} \\
 & & \ldots & \\
 & | & 9 & \{\ \mathrm{print}('9')\ \}
\end{array}
$$

$$A \rightarrow A\alpha \mid A\beta \mid \gamma$$

$$
\begin{array}{lcl}
A & \rightarrow & \gamma R \\
R & \rightarrow & \alpha R \mid \beta R \mid \epsilon
\end{array}
$$

$$
\begin{array}{lcl}
A & = & expr \\
\alpha & = & + \ term \ \{\ \mathrm{print}('+')\ \} \\
\beta & = & - \ term \ \{\ \mathrm{print}('-')\ \} \\
\gamma & = & term
\end{array}
$$

Let's Build A Translator: Arithmetic Expressions to Postfix

$$
\begin{aligned}
expr &\rightarrow term\ rest \\
rest &\rightarrow +\ term\ \{\ \mathrm{print}('+')\ \}\ rest \\
&\mid\ -\ term\ \{\ \mathrm{print}('-')\ \}\ rest \\
&\mid\ \epsilon \\
term &\rightarrow 0\ \{\ \mathrm{print}('0')\ \} \\
&\mid\ 1\ \{\ \mathrm{print}('1')\ \} \\
&\ \ \ \ \dots \\
&\mid\ 9\ \{\ \mathrm{print}('9')\ \}
\end{aligned}
$$

$$A \rightarrow A\alpha \mid A\beta \mid \gamma$$

$$
\begin{aligned}
A &\rightarrow \gamma R \\
R &\rightarrow \alpha R \mid \beta R \mid \epsilon
\end{aligned}
$$

$$
\begin{aligned}
A &= expr \\
\alpha &= +\ term\ \{\ \mathrm{print}('+')\ \} \\
\beta &= -\ term\ \{\ \mathrm{print}('-')\ \} \\
\gamma &= term
\end{aligned}
$$

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

$$
\begin{array}{lcll}
\mathit{expr} & \rightarrow & \mathit{expr} + \mathit{term} & \{\ \mathrm{print}('+')\ \} \\
 & | & \mathit{expr} - \mathit{term} & \{\ \mathrm{print}('-')\ \} \\
 & | & \mathit{term} & \\
\\
\mathit{term} & \rightarrow & \mathit{term} * \mathit{factor} & \{\ \mathrm{print}('*')\ \} \\
 & | & \mathit{term}\ /\ \mathit{factor} & \{\ \mathrm{print}('/')\ \} \\
 & | & \mathit{factor} & \\
\\
\mathit{factor} & \rightarrow & (\ \mathit{expr}\ ) & \\
 & | & \mathbf{num} & \{\ \mathrm{print}(\mathbf{num}.\mathit{value})\ \} \\
 & | & \mathbf{id} & \{\ \mathrm{print}(\mathbf{id}.\mathit{lexeme})\ \}
\end{array}
$$

$$
\begin{array}{lcll}
expr & \rightarrow & expr + term & \{\ \text{print}('+')\ \} \\
 & | & expr - term & \{\ \text{print}('-')\ \} \\
 & | & term & \\
 & & & \\
term & \rightarrow & term * factor & \{\ \text{print}('*')\ \} \\
 & | & term\ /\ factor & \{\ \text{print}('/')\ \} \\
 & | & factor & \\
 & & & \\
factor & \rightarrow & (\ expr\ ) & \\
 & | & \mathbf{num} & \{\ \text{print}(\mathbf{num}.value)\ \} \\
 & | & \mathbf{id} & \{\ \text{print}(\mathbf{id}.lexeme)\ \}
\end{array}
$$

Thanks to the lexical analyzer
the parser can deal with identifier

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

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

$$31 + 28 + 59 \Rightarrow \langle \mathbf{num}, 31 \rangle \ \langle + \rangle \ \langle \mathbf{num}, 28 \rangle \ \langle + \rangle \ \langle \mathbf{num}, 59 \rangle$$

```
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"⟩ ⟨=⟩ ⟨**id**, "count"⟩ ⟨+⟩ ⟨**id**, "increment"⟩ ⟨;⟩

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

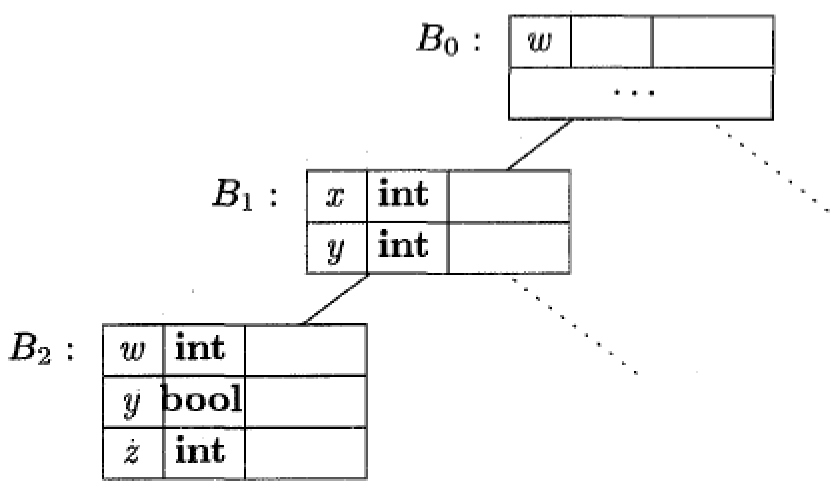
The string table

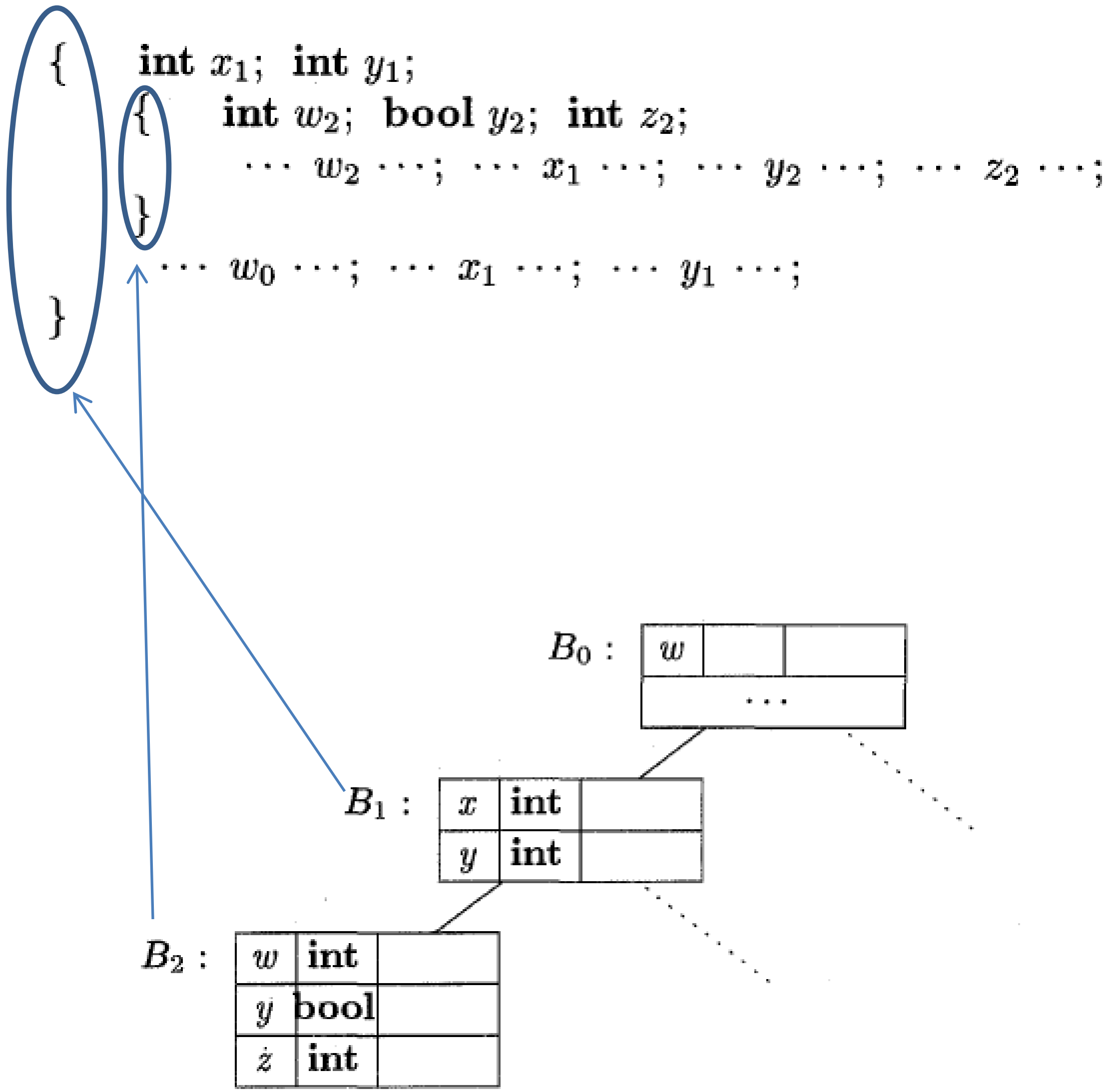
```
if ( peek holds a letter ) {
        collect letters or digits into a buffer b;
        s = string formed from the characters in b;
        w = token returned by words.get(s);
        if ( w is not null ) return w;
        else {
                Enter the key-value pair (s, ⟨id, s⟩) into words
                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 x1; int y1;
{ int w2; bool y2; int z2;
··· w2 ···; ··· x1 ···; ··· y2 ···; ··· z2 ···;
}
··· w0 ···; ··· x1 ···; ··· y1 ···;
}
B0 :
w
···
B1 :
x int
y int
B2 :
w int
y bool
z int

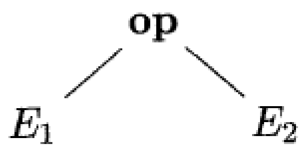
How Are Symbol Tables Accessed?

- Using semantic action
- A semantic action can put information in symbol table
- A semantic action can get information from symbol table

```
program  →                        { top = null; }
                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?