CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

- **A Translator for Simple Expressions**
- **Lexical Analysis**

A Translator for Simple Expressions

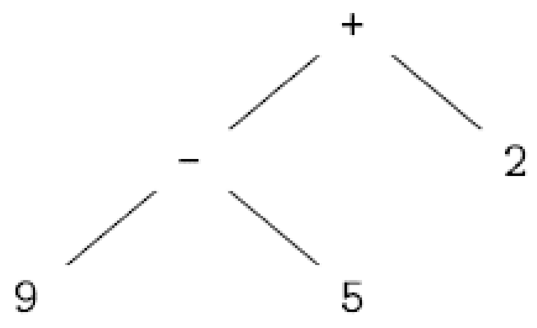
- Objective:
  - In this example, we need to do infix to postfix translation
  - We start with just plus and minus
  - Finding a grammar for the desired language is one problem
  - Given a grammar, constructing a translator for the language is another problem
  - We are tackling the second problem.
  - One problem we must solve is that,

  this grammar is left recursive.

```
expr → expr + term { print('+') }
expr → expr - term { print('-') }
expr → term
term → 0           { print('0') }
. . .
term → 9           { print('9') }
```

A Translator for Simple Expressions

- **Abstract and Concrete Syntax**
  - If the syntax tree would just have the operators + and - and the 10 digits 0,1,...,9. That would be called the abstract syntax tree. (AST)
  - A parse tree coming from a grammar is technically called a concrete syntax tree.
  - An (AST) example: input is “9-5+2”.
  - The root represents the operator +.



A Translator for Simple Expressions

- **Adapting the Translation Scheme**
  - From the last time of the class, we have this:
    - In general, for any nonterminal A, and any strings α, and β (α and ß cannot start with A). we can replace the pair of productions in this way!

$$A \to A\ \alpha \mid \beta$$

with the triple

$$A \to \beta\ R$$
$$R \to \alpha\ R \mid \varepsilon$$

New Form

$$A \to \beta\ R$$
$$R \to \alpha\ R \mid \varepsilon$$

Original

```
expr → term rest
rest → + term rest
rest → ε
```

    - Where A is expr, **R is rest**, **α is + term**, and β is term.
    - And we find there is a mapping to our original production and removed the left recursion!!

A Translator for Simple Expressions

- **Adapting the Translation Scheme (Cont.)**
  - This time there are two operators + and - so we replace the triple
    - A → A α | A β | γ

  with the quadruple
    - A → γ R
    - R → α R | β R | ε

  And this time we have "actions", check this:

```
expr → expr + term { print('+') }
expr → expr - term { print('-') }
expr → term
term → 0           { print('0') }
. . .
term → 9           { print('9') }
```

    - Similarly, α is + term { print('+') } ; β is - term { print('-') }; γ is term; A is expr; and **R is rest**
  - Eventually, the formulas still hold and we get
  - It is said that the left recursion is removed!
  - This is the translation scheme for

  Left-recursion elimination

```
expr → term rest
rest → + term { print('+') } rest
     | - term { print('-') } rest
     | ε
term → 0           { print('0') }
. . .
     | 9           { print('9') }
```

A Translator for Simple Expressions

- **Adapting the Translation Scheme (Cont.)**
  - **Finally, we translate 9-5+2 to 95-2+**

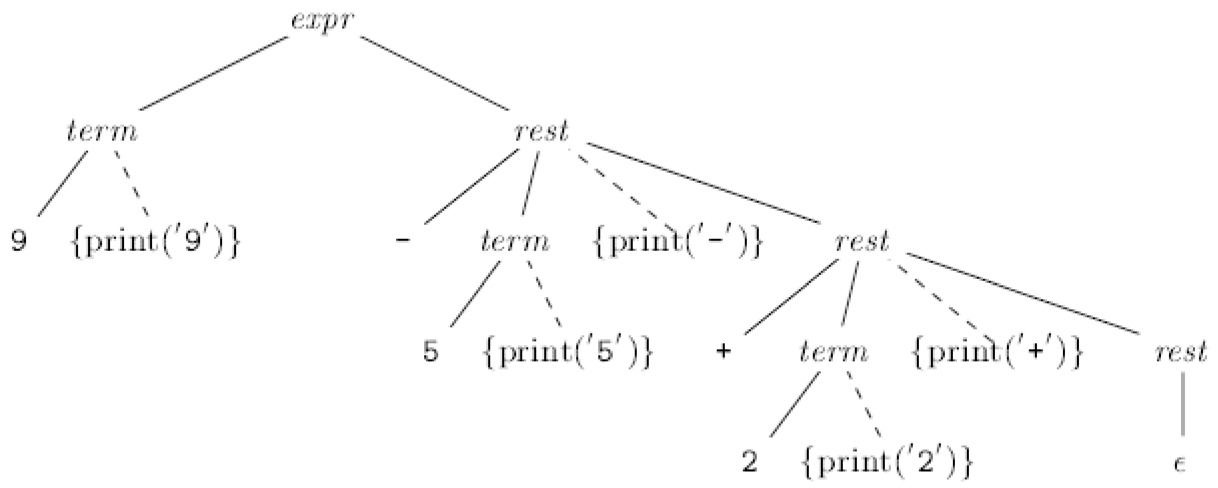


Figure 2.24: Translation of 9-5+2 to 95-2+

A Translator for Simple Expressions

- **Procedures for the Non-terminals *expr*, *term*, and *rest***
  - Fig. 2.25 implement the syntax-directed translation scheme in Fig. 2.23.
  - These functions mimic the production bodies of the corresponding non-terminals.
  - expr implements the production

  expr → term rest,

  by the calls term(),

  followed by rest().
  - See the next page for

  Fig. 2.25 pseudo-code

$$
\begin{array}{rcl}
expr & \rightarrow & term\ rest \\[6pt]
rest & \rightarrow & +\ term\ \{\ \mathrm{print}('+')\ \}\ \ rest \\
 & | & -\ term\ \{\ \mathrm{print}('-')\ \}\ \ rest \\
 & | & \epsilon \\[6pt]
term & \rightarrow & 0\ \ \{\ \mathrm{print}('0')\ \} \\
 & | & 1\ \ \{\ \mathrm{print}('1')\ \} \\
 & & \cdots \\
 & | & 9\ \ \{\ \mathrm{print}('9')\ \}
\end{array}
$$

Figure 2.23: Translation scheme after left-recursion elimination

A Translator for Simple Expressions

```
void expr() {
        term(); rest();
}

void rest() {
        if ( lookahead == '+' ) {
                match('+'); term(); print('+'); rest();
        }
        else if ( lookahead == '-' ) {
                match('-'); term(); print('-'); rest();
        }
        else { } /* do nothing with the input */ ;
}

void term() {
        if ( lookahead is a digit ) {
                t = lookahead; match(lookahead); print(t);
        }
        else report("syntax error");
}
```

Figure 2.25: Pseudocode for nonterminals *expr*, *rest*, and *term*.

- If the test succeeds, variable t saves the digit represented by lookahead so it can be written after the call to match.
- Note that match changes the lookahead symbol, so the digit needs to be saved for later printing

rest→ε

The epsilon production is only used when all others fail (that is why it is the else arm and not the other else if arms).

A Translator for Simple Expressions

- **Simplifying the translator**

Our job is to eliminate the tail recursive. How to do it?

Next page is the solution!

Tail recursive!

```
void expr() {
        term(); rest();
}

void rest() {
        if ( lookahead == '+' ) {
                match('+'); term(); print('+'); rest();
        }
        else if ( lookahead == '-' ) {
                match('-'); term(); print('-'); rest();
        }
        else { } /* do nothing with the input */ ;
}

void term() {
        if ( lookahead is a digit ) {
                t = lookahead; match(lookahead); print(t);
        }
        else report("syntax error");
}
```

Figure 2.25: Pseudocode for nonterminals *expr*, *rest*, and *term*.

A Translator for Simple Expressions

```
void rest() {
        while( true ) {
                if( lookahead == '+' ) {
                        match('+'); term(); print('+'); continue;
                }
                else if ( lookahead == '-' ) {
                        match('-'); term(); print('-'); continue;
                }
                break ;
        }
}
```

A Translator for Simple Expressions

- **The complete program**
  - The complete parser program which is written in Java
  - Infix to postfix
  - Very useful. In chapter 2.5.5

Lexical Analysis

- The purpose of lexical analysis is to convert: a sequence of characters (the source) $\rightarrow$ a sequence of tokens
- How? Input characters are becoming “token objects” by the grouping
- In this section, a token is (a terminal + additional information)
- The lexical analyzer in this section allows numbers, identiers, and “white space" (blanks, tabs, and newlines) to appear within expressions

Lexical Analysis

- Now we extend the original little bit by incorporating * and /

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

Figure 2.28: Actions for translating into postfix notation

Lexical Analysis

- The lexer operates on the **input** and the resulting token sequence, is the input to the **parser**.
  - Remember the 2 big boxes and several stages?
- The reason we were able to produce the translator in the previous section without a lexer is that, all the tokens were just one character (that is why we had just single digits).

Lexical Analysis

- Removal of White Space and Comments
  - These do not become tokens so that the parser need not worry about them.
  - C-like pseudo code

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;
}

Figure 2.29: Skipping white space

Lexical Analysis

- Reading Ahead
  - Consider distinguishing x<y from x<=y.
  - After reading the “<“ we must read another character.
    - If it is y, we have found our token (<).
    - However, we must unread the y so that when asked for the next token, we will start at y. If it is never more than one extra character that must be examined, a single char variable would suffice. A more general solution is discussed in the next chapter (Lexical Analysis).
    - A lexical analyzer for C or Java must read ahead after it sees the character >. If the next character is =, then “>” is just “part” of the character sequence “>=“, the lexeme for “>=”
    - A general approach to reading ahead on the input, is to maintain an input buffer from which the lexical analyzer can read characters
    - For Input buffers, since fetching a block of characters is usually more efficient than fetching one character at a time

Lexical Analysis

- **Constants**
  - About the ways in dealing with constants in lexical analysis
  - Integer constants can be allowed either by creating a terminal symbol, say num, for such constants or by incorporating the syntax of integer constants into the grammar
  - In the book, there is a pseudo code:
    - Figure 2.30: Grouping digits into integers

Lexical Analysis

- **Recognizing identifiers and keywords**
    - <Ch 2.6.4> TBD