CSE 504 Compiler Design
A Simple Compiler (1)

YoungMin Kwon

Simple Compiler: Objective

- Learn the overall phases of a compiler
- Learn how to write a grammar
- Translate a source code to an abstract stack machine code
    - Lexical scanner
    - Parser
    - Code generation
- Learn abstract stack machines

Syntax Definition

- Context-Free Grammars
  - Naturally describe the hierarchical structure of many programming languages
- e.g. if-else statement in C language
  - `if ( expression ) statement else statement`
  - In the context-free grammar
  - `stmt -> IF ( expr ) stmt ELSE stmt`,
  - where `stmt` and `expr` are nonterminals representing statements and expressions
    `IF`, `ELSE`, `(`, and `)` are tokens
  - Such rules are called a production and `->` may be read as “can have the form”

Context-Free Grammar

- 4 Components
  1. A set of tokens (terminals)
  2. A set of nonterminals
  3. A set of productions composed of
     - left side: a nonterminal
     - arrow: `->`
     - right side: a sequence of terminals and nonterminals
  4. A start symbol (first production is for the start symbol)
- Productions with the same left side can be grouped (separated by `|`)

Context-Free Grammar (Example)

- Example
  - ```
    list -> list + digit
    list -> list – digit
    list -> digit
    digit -> 0 | 1 | 2 | … | 9
    ```
  - ```
    string -> string + string
           |  string – string
           |  digit
    digit -> 0 | 1 | 2 | … | 9
    ```

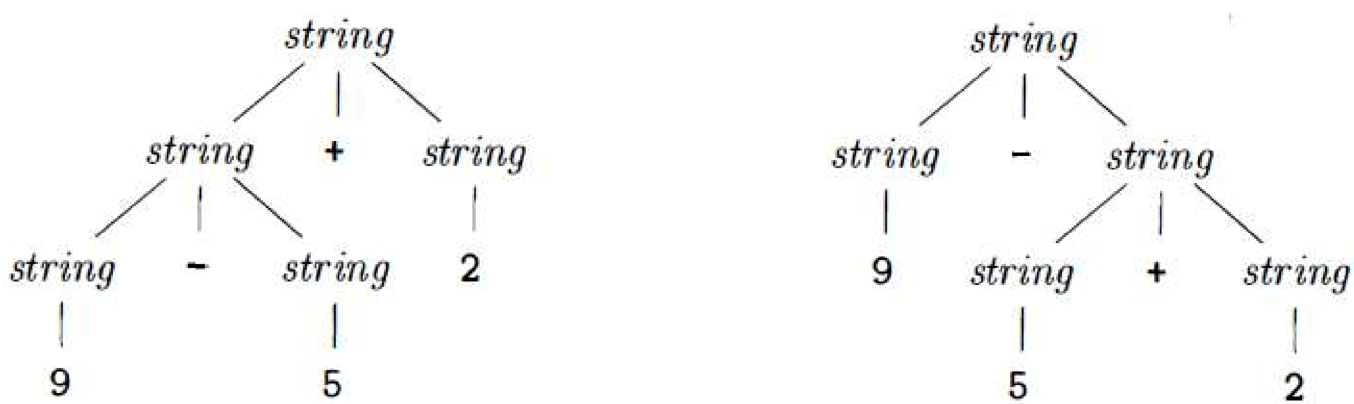
Context-Free Grammar: Derivations and Language

- A grammar derives strings by beginning with the start symbol and repeatedly replacing the nonterminals with the body of the corresponding production.
- All terminal strings derived from the start symbol form the language defined by the grammar.
- e.g. we can deduce that 9-5+2 is a list as follows
  - 9, 5, 2 are digits by the productions `digit -> 9`, `digit -> 5`, `digit -> 2`
  - 9 is a list by the production `list -> digit` (9 is a digit)
  - 9-5 is a list by the production `list -> list - digit` (9 is a list, 5 is a digit)
  - 9-5+2 is a list by the production `list -> list + digit` (9-5 is a list, 5 is a digit)
- Parsing is the process of finding the deduction tree for a grammar from a terminal string.

Context-Free Grammar (Ambiguity)

- The grammar for string is ambiguous.
- e.g. Two parse trees for 9 - 5 + 2
  - (9 - 5) + 2 and 9 - (5 + 2)

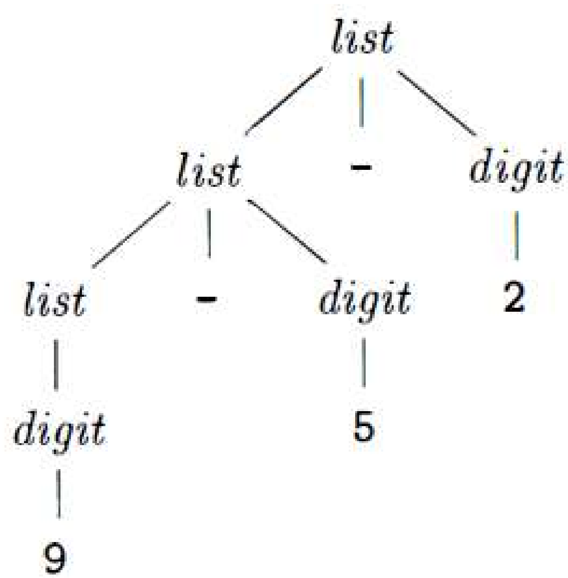
```
string -> string + string
        |  string - string
        |  digit
 digit -> 0 | 1 | 2 | … | 9
```

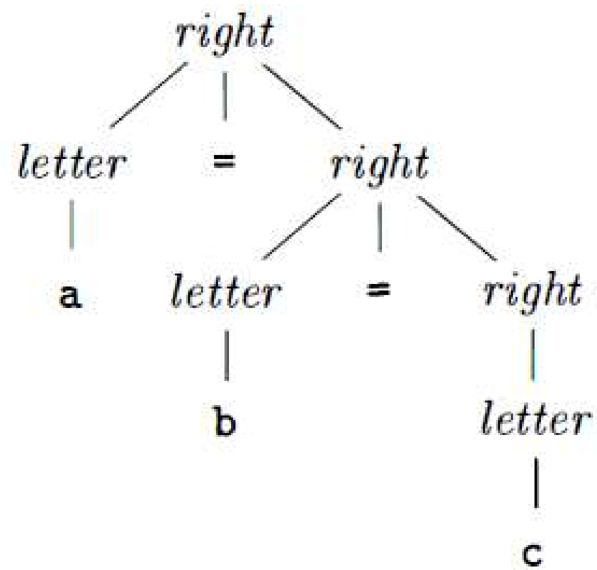


Associativity to fix the ambiguity

- Left associativity:
  - `9 – 5 + 2` should be read as `(9 – 5) + 2`
  - ```
    list -> list + digit
          |  list – digit
          |  digit
    ```
  - If 5 + 2 became a list first, there are no productions that can derive further.
- Right associativity:
  - `a = b = c` should be read as `a = (b = c)`
  - ```
     right -> letter = right
            |  letter
    letter -> a | b | … | z
    ```
  - If a = b became a right first, there are no productions that can derive further.

Parse trees for 9 - 5 + 2 and a = b = c





Precedence to fix the ambiguity

- Precedence
  - 1 + 2 * 3 should be read as 1 + (2 * 3) not (1 + 2) * 3
- To fix the precedence, we can add a new nonterminal `term`

```
expr -> expr + term
     |  expr - term
     |  term
term -> term * digit
     |  term / digit
     |  digit
digit -> 0 | 1 | … | 9
```

- Observe that if 1 + 2 became an expr first, we cannot build a parse tree: there are no productions like `expr -> expr * term`

Simple Compiler: Syntax for expressions

```
• expr -> expr + term
        |  expr - term
        |  term
• term -> term * factor
        |  term / factor
        |  factor
• factor -> NUMBER
          |  IDENTIFIER
          |  ( expr )
```

- Quiz: with the context-free grammar above, build a parse tree for x - 2 * (3 + y)

Simple Compiler: Syntax for statements

```
stmt -> ID := expr ;
      |  IF expr THEN stmt
      |  IF expr THEN stmt ELSE stmt
      |  WHILE expr DO stmt
      |  BEGIN opt_stmts END
opt_stmts -> ε
           |  opt_stmts stmt
```

- Quiz: with the context-free grammar above, build a parse tree for

```
IF x
THEN
   x := 0;
ELSE
   BEGIN
      y := y + 1;
      x := 1;
   END
```

Syntax-Directed Definition

- Specifies the translation of a construct in terms of attributes associated with its syntactic components
  1. Associate a set of attributes to each grammar symbol
     - E.g. attributes: type, memory location of a code, string …
  2. Add a set of semantic rules for computing values of the attributes associated with the symbols in the production
- Types of Attributes:
  - Inherited attributes: attributes that are dependent on it's parent, sibling, and self nodes
  - Synthesized attributes: attributes that are dependent on it's child and self nodes.

Postfix Notation

- Postfix notation of an expression E can be inductively defined as
  - If E is a variable or a constant, postfix notation of E is E itself
  - If E is an expression of the form E_1 op E_2, the postfix notation of E is E'_1 E'_2 op, where E'_1 and E'_2 are the postfix notations for E_1 and E_2
  - If E is of the form (E_1), the postfix notation of E is the postfix notation of E_1
- e.g. the postfix notation of (9-5)+2 is 95-2+
- To evaluate the postfix notation, repeatedly find the left most operator and replace the operator and the two numbers on its right with their evaluation.
- e.g. 95-2+ -> 42+ -> 6
- Quiz: evaluate the postfix notation 952+-3*

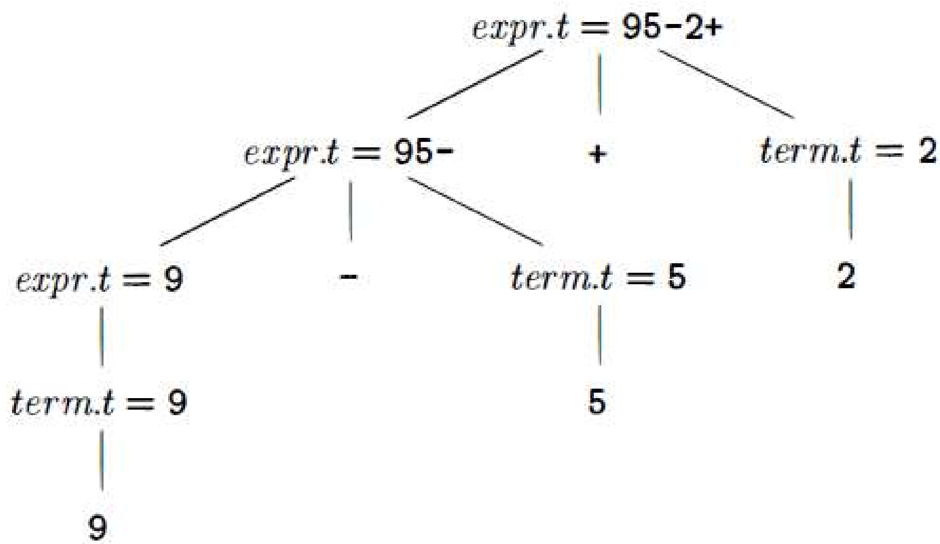
Syntax-Directed Definition for infix to postfix translation

Production	**Semantic Rule**
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t$ | $term.t$ | '+'
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t$ | $term.t$ | '-'
expr -> term	expr.t = term.t
term -> 0	term.t = '0'
term -> 1	term.t = '1'
…	…
term -> 9	term.t = '9'

where | means the string concatenation.

Syntax-Directed Definition for infix to postfix translation

- Example attributes for 9 – 5 + 2

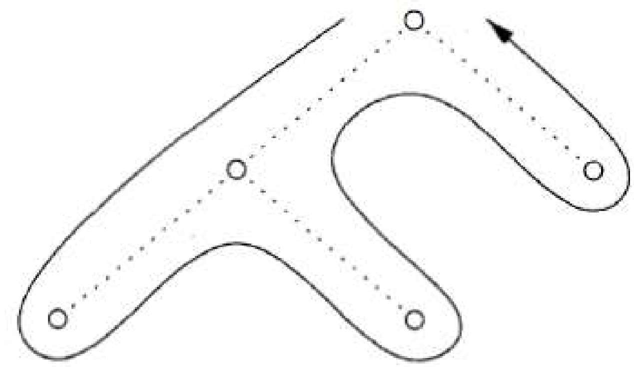


- Quiz: Update the syntax-directed definition with `factor` and compute the attributes of 1 - 2 * 3 + 4

Syntax-Directed Definition: tree traversal

- One way to compute the attributes is to travers the parse tree in the depth first manner.
- Depth first traversal

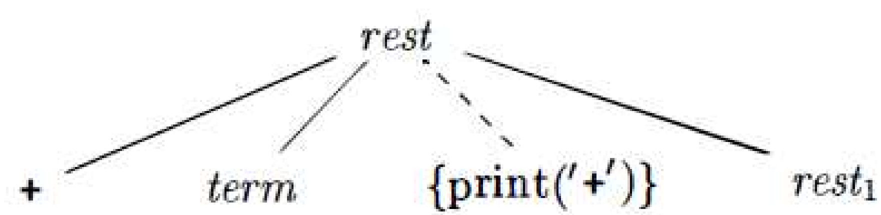
```
procedure visit(node N) {
   foreach child C of N, from left to right {
      visit(C);
   }
   evaluate semantic rules at node N;
}
```



- The picture on the right is an example of depth first traversal
- Check how the attributes in the parse tree (9-5+2) of the previous page is computed by the depth first traversal.

Translation Scheme

- Definition: translation Scheme is a context-free grammar in which program fragments called semantic actions are embedded within the right sides of productions
- Translation Scheme is an alternative way of translation without manipulating strings.
  - If we perform the semantic actions as we encounter them while depth first traversing the tree, we can produce the same postfix translation.
- Example
  - `rest -> + term { print('+') } rest`$_1$
- The parse tree below shows an extra leaf from the semantic action

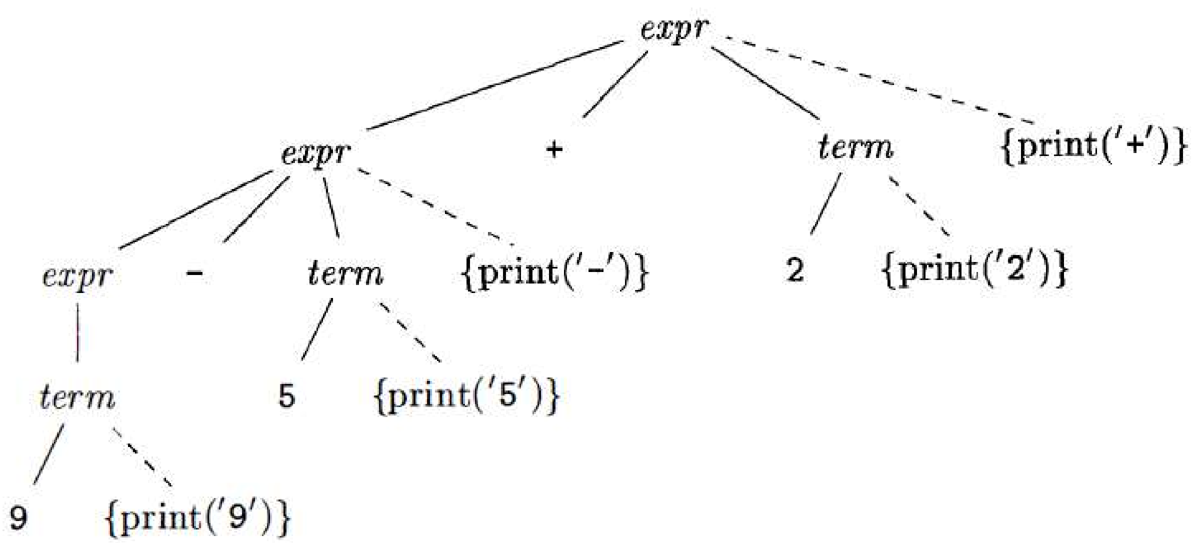


Translation Scheme

- Actions for infix to postfix translation

$expr$	$\rightarrow$	$expr_1$ + $term$	{print('+')}
$expr$	$\rightarrow$	$expr_1$ - $term$	{print('-')}
$expr$	$\rightarrow$	$term$	
$term$	$\rightarrow$	0	{print('0')}
$term$	$\rightarrow$	1	{print('1')}
		...	
$term$	$\rightarrow$	9	{print('9')}

- Actions translating 9+5-2 into 95-2+



- Quiz: Update the translation scheme with `factor` and check how 1 - 2 * 3 + 4 is translated into a postfix notation.