CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

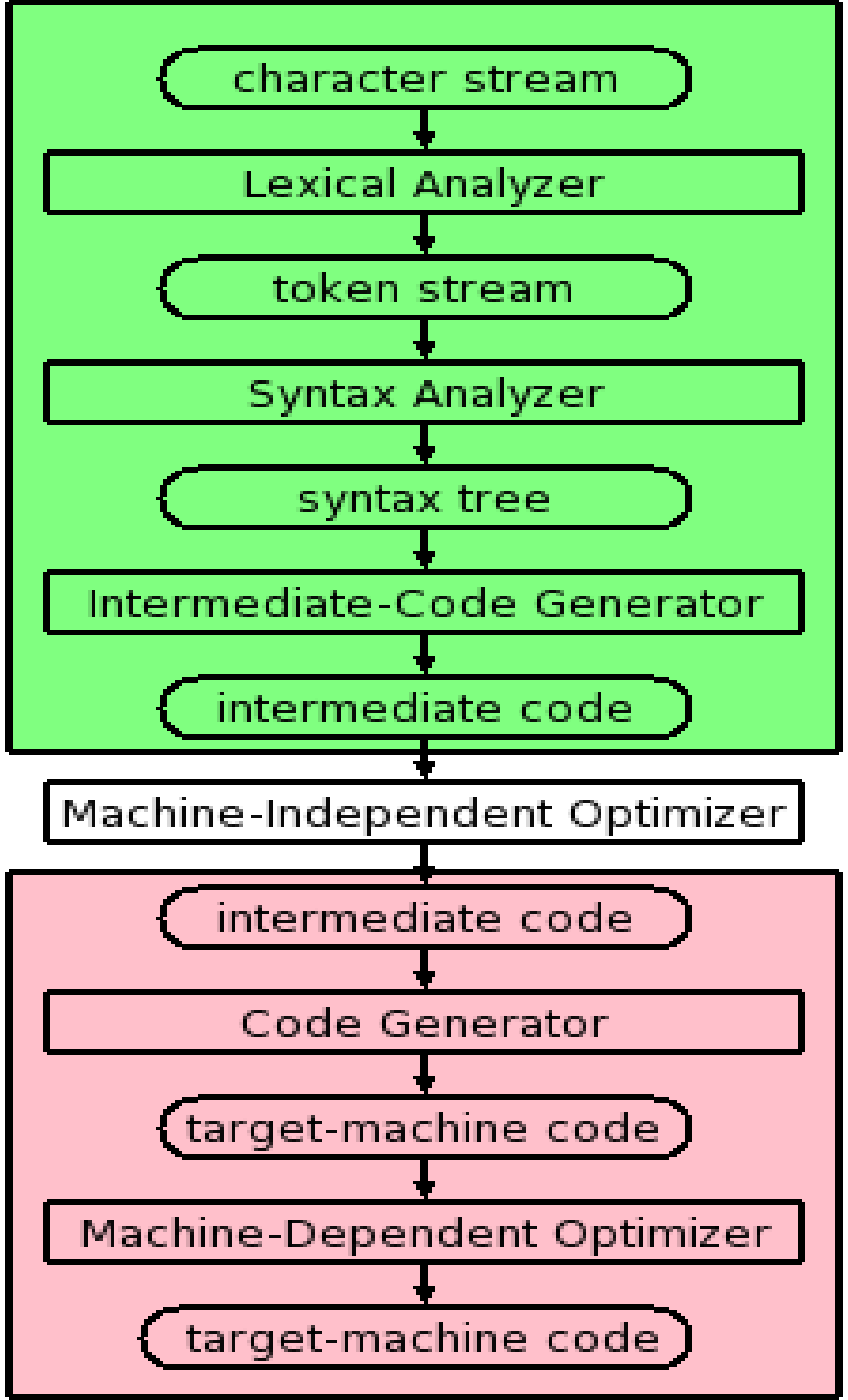
- The goal of this chapter is to know the idea

of a very simple compiler.

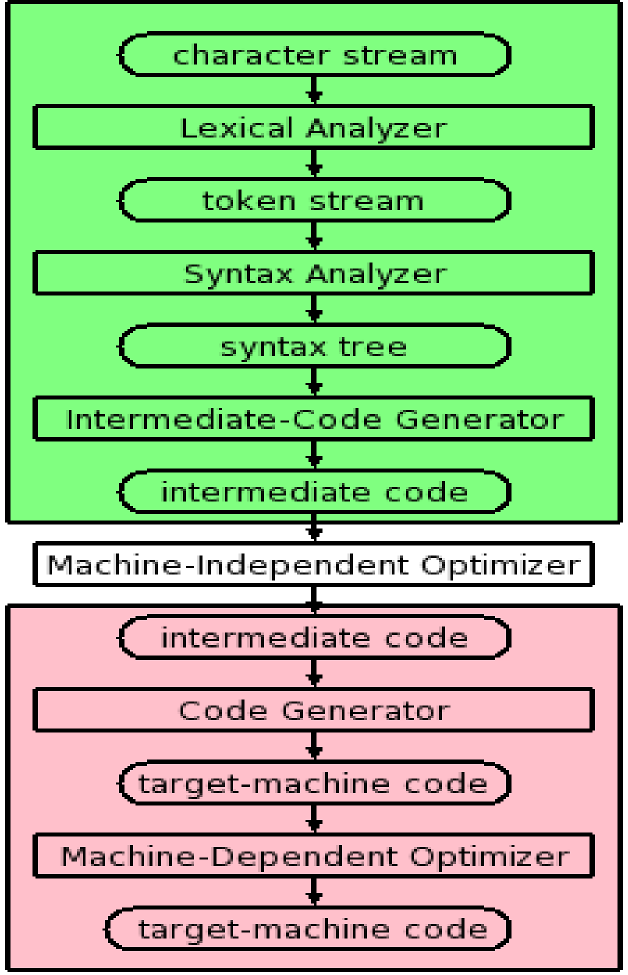
- Really we are just going as far as the interme end. (i.e. front end)
- Nonetheless, the output,

i.e. the intermediate code,

does look somewhat like assembly language.



- The Structure of a Compiler
  - Lexical Analysis (or Scanning)
  - Syntax analysis (parsing)
  - Semantic Analysis
  - Intermediate code generation
  - Code optimization
  - Code generation
  - Symbol-Table Management (not specifically pointed in book)
  - Error Handling Routing (not specifically pointed out in book)



- In this chapter, there is:
  - A simple source language.
  - A target close to the source.
  - No optimization.
  - No machine-dependent back end.
  - No tools.
  - Little theory.
- A quick review:
  - What is the job of Lexical Analysis ? Scanning or Tokenizing
  - What is the job of Syntax Analysis ? Parsing
  - This is Ch.2.
    - Ch.3 is the Lexical Analysis and Ch.4 is the Syntax Analysis
  - We are pretty much on the warm-up stage for Ch.4 in this chapter

Introduction

- The *syntax* describes the *form* of a program in a given language.
- The *semantics* describes the *meaning* of that program.

(They are different!)

- We will learn the standard representation for the syntax
  - *Context-free grammars* also called BNF (Backus-Naur Form).

Syntax Definition

- **Definition of Grammars**
  - A *context-free grammar* (CFG) consists of
    - A set of *terminals*
    - A set of *non-terminals.*
    - A set of *productions* (**rules** for transforming non-terminals). This is written as LHS → RHS, where LHS is a single non-terminal
      - The RHS is a string containing non-terminals and/or terminals.
    - A specific nonterminal designated as start symbol

Syntax Definition

- **Definition of Grammars (Cont.)**

Example:

```
Terminals: 0 1 2 3 4 5 6 7 8 9 + -
Nonterminals: list digit
Productions:
    list → list + digit
    list → list - digit
    list → digit
    digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Start symbol: list
```

```
A → B | C
```

is simply shorthand for

```
A → B
A → C
```

Syntax Definition

- If no start symbol is specifically designated, the **LHS** of the first production is the start symbol.

- **Derivations**
  - This is the process of applying productions, starting with the start symbol and ending when only terminals are present is called a *derivation*
  - The final string has been *derived* from the initial string (in this case the start symbol).
  - See how we generate 7+4-5
    - It begins with the start symbol, applying productions, and stopping when no productions can be applied (because only terminals remain).
    - See the example in the next page

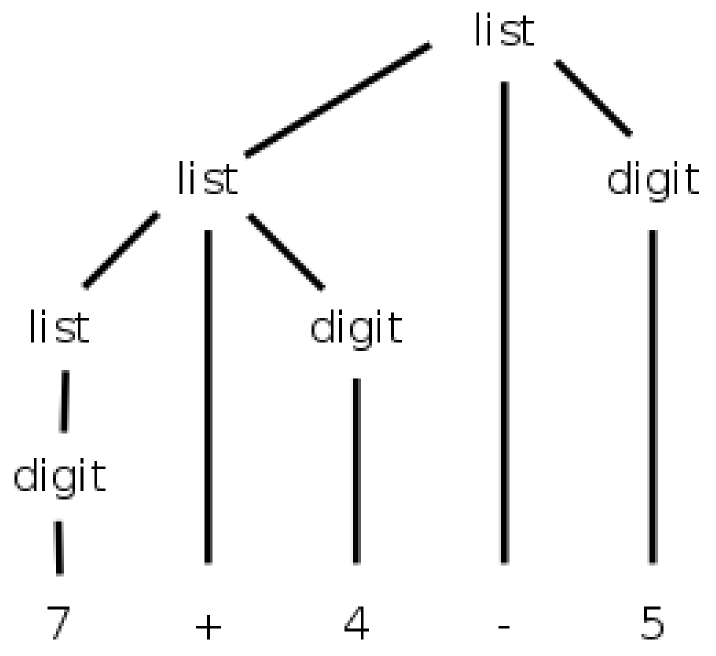
Syntax Definition

```
list → list - digit
     → list - 5
     → list + digit - 5
     → list + 4 - 5
     → digit + 4 - 5
     → 7 + 4 - 5
```

- The set of all strings derivable from the **start** symbol is the *language generated by the CFG (Yes! This is my name!)*
- The way you get different final expressions is that you make different choices of which production to apply. There are **3 productions you can apply to list** and **10 ways you can apply to digit**. (The derivation you had seen is ONLY one of the them!)
- The result cannot have blanks since blank is not a terminal.
- The empty string is not possible since, starting from list, we cannot get to the empty string. If we wanted to include the empty string, we would add the production, list → ε
- **Q&A: If there is a number, say 25, allowed as an operand in the process of derivation?**

Syntax Definition

- Given a grammar, *parsing* a string (of terminals) consists of determining if the string is in the language generated by the grammar.
  - If it is in the language, parsing produces a derivation.
  - If it is not, parsing reports an error.
- **Parse Trees**
  - Our previous example, 7+4-5
  - Parse Tree is constructed
  - The leaves of the tree, read from left to right,
  is called the *yield* of the tree.
  - We say that this string is *derived,*
  from the (nonterminal at the) root,
  or is *generated* by the root



Syntax Definition

- **Ambiguity**
  - An *ambiguous* grammar may used to construct 2 or more parse trees
  - And these can yield to the SAME final string
  - Avoid such grammars when designing a new programming language

Syntax Definition

- **Ambiguity (Cont.)** --- An example of erroneously prove of ambiguity
  - Consider the grammar

S → A B

A → x

B → x

  - And if you say the grammar is ambiguous because we can derive the string:

x x in two ways. It is WRONG! They have the same parse tree!

S → A B → A x → x x

S → A B → x B → x x

Syntax Definition

l → l - d
→ l - 8
→ l - d - 8
→ l - 8 - 8
→ d - 8 - 8
→ 8 - 8 - 8

- **Associativity of operators**
  - Our grammar gives left associativity
  - If you traverse the parse tree in post-order and perform the indicated arithmetic you will evaluate the string left to right.
  - 8-8-8, this can be correctly derived using our grammar and is evaluated to -8
  - What if, we replace the first 2 productions? New Grammar!

```
list → digit + list
list → digit - list
```

```
Terminals: 0 1 2 3 4 5 6 7 8 9 + -
Nonterminals: list digit
Productions:
    list → list + digit
    list → list - digit
    list → digit
    digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Start symbol: list
```

  - It will be right associative!

Syntax Definition

- **Precedence of operators**
  - We normally want * to have higher precedence than +.
  - We do this by introducing an additional nonterminal to indicate the items that have been multiplied. (in this grammar, it is “term”!)
  - The example below gives the four basic arithmetic operations their normal precedence unless overridden by parentheses.

```
expr   → expr + term | expr - term | term
term   → term * factor | term / factor | factor
factor → digit | ( expr )
digit  → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Syntax Definition

- In the quiz,
  - I might like to ask you to generate a parsing tree according to some grammar.
  - Or, I give you a string (i.e. a-b+c) and a grammar. Then, I ask you if this is possible?

Syntax-Directed Translation

- Postfix Notation
  - This notation is called postfix because the rule is operator after operand(s).
  - The notation we normally use is called infix because the rules is operator in between operands
  - For example, E op F becomes **E' F' op**, where E' and F' are the postfix of E and F respectively.
  - Our question is, given, say 1+2-3, what are E, F and op? Does E=1+2, F=3, and op=-? Or does E=1, F=2-3 and op=+? This is the issue of precedence and associativity mentioned above.
  - To simplify the present discussion we will start with fully parenthesized infix expressions.

Syntax-Directed Translation

- Example1
  - 1+2/3-4*5
  - Parenthesize (using standard precedence) to get (1+(2/3))-(4*5)
  - Apply the rule mentioned earlier (still remember “E’ F’ op” ?) to calculate

  P{(1+(2/3))-(4*5)}, where P{X} means “the function” to convert the infix expression X to postfix
  - Step A~ F

A. P{(1+(2/3))-(4*5)}
B. P{(1+(2/3))} P{(4*5)} -
C. P{1+(2/3)} P{4*5} -
D. P{1} P{2/3} + P{4} P{5} * -
E. 1 P{2} P{3} / + 4 5 * -
F. 1 2 3 / + 4 5 * -

Syntax-Directed Translation

- Example2
  - (1+2)/3-4*5
  - Parenthesize to get ((1+2)/3)-(4*5)
  - Calculate P{((1+2)/3)-(4*5)}
  - Step A~ F

A. P{((1+2)/3) P{(4*5)} -
B. P{(1+2)/3} P{4*5) -
C. P{(1+2)} P{3} / P{4} P{5} *
D. P{1+2} 3 / 4 5 * -
E. P{1} P{2} + 3 / 4 5 * -
F. 1 2 + 3 / 4 5 * -

Syntax-Directed Translation

- The example1 and example2 is a way to construct the tree in the bottom-up style!
- Bottom: infix
- Up: postfix
- This is the “translation”

Syntax-Directed Translation

- Synthesized Attributes
  - We want to decorate the parse trees we construct with annotations that give the value of certain attributes of the corresponding node of the tree.
  - For convenience, the grammar is repeated just below.
  - This grammar supports parentheses, although our example 1+2/3-4*5 does not use them

```
expr   → expr + term | expr - term | term
term   → term * factor | term / factor | factor
factor → digit | ( expr )
digit  → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Syntax-Directed Translation

- An attribute is said to be synthesized if its value at a parse-tree node N is determined from attribute values at the children of N and N itself.
- Synthesized attributes have the desirable property that they can be evaluated during a single bottom-up traversal of a parse tree.

Syntax-Directed Translation

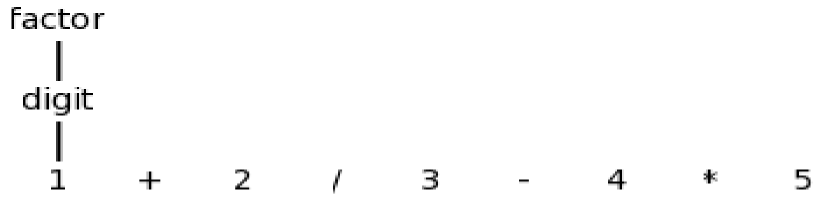
- Here is a “movie” which a parse tree is built from the example:
- 1+2/3-4*5

```
expr   → expr + term | expr - term | term
term   → term * factor | term / factor | factor
factor → digit | ( expr )
digit  → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

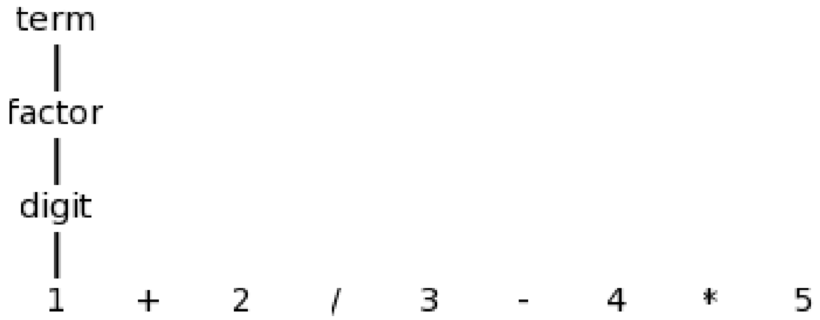
Syntax-Directed Translation

- Step1:

- Step2:

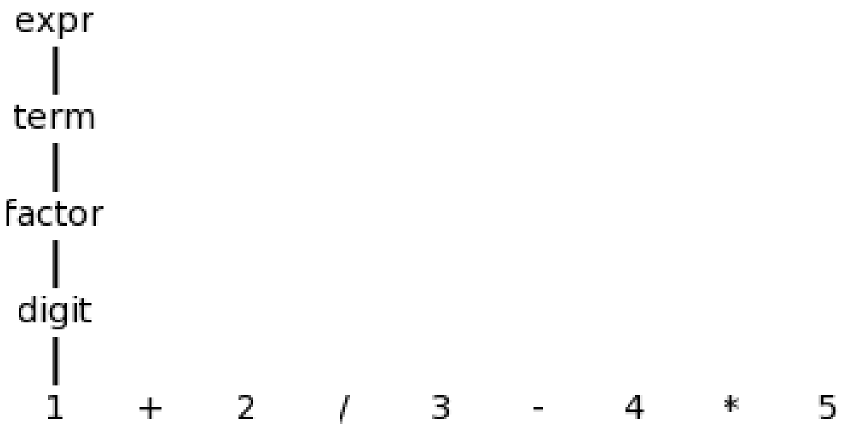


- Step3:

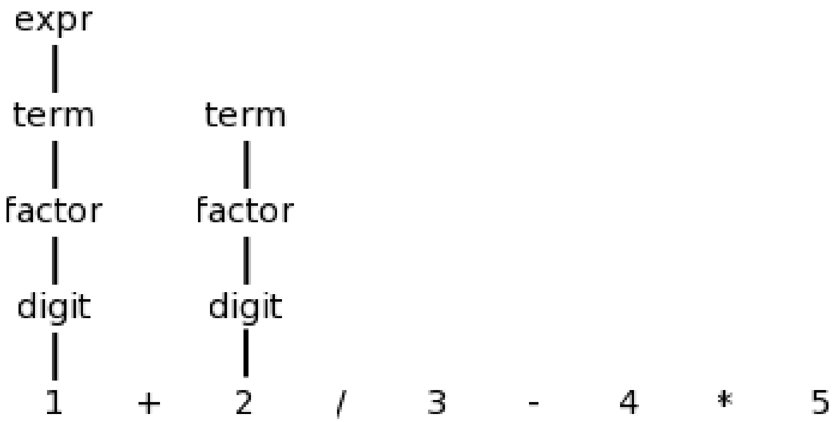


Syntax-Directed Translation

- Step4:

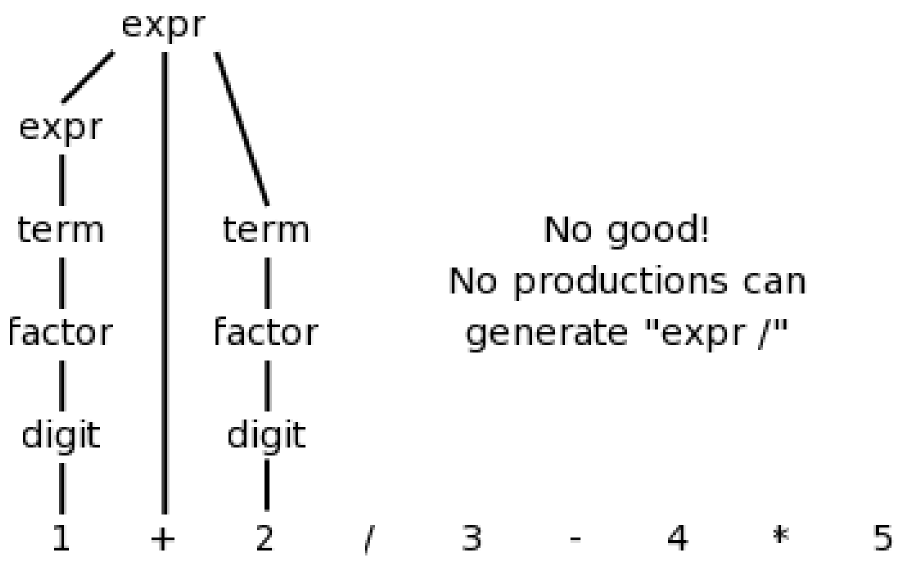


- Step5:



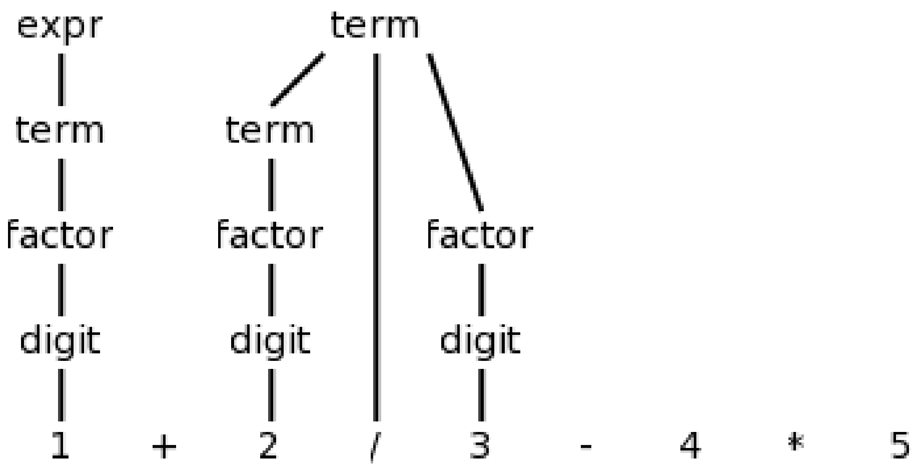
Syntax-Directed Translation

- Step6:



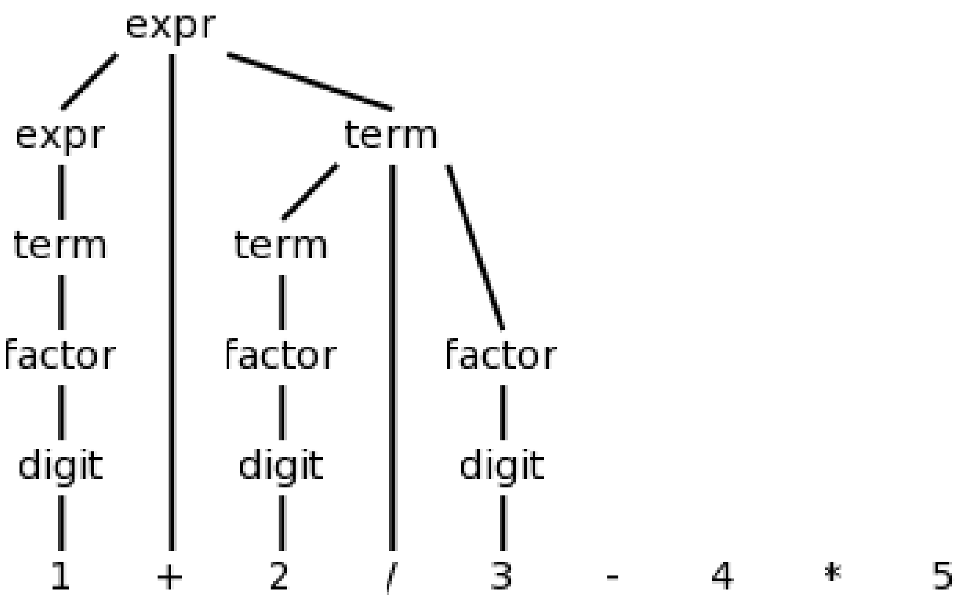
Syntax-Directed Translation

- Step7:



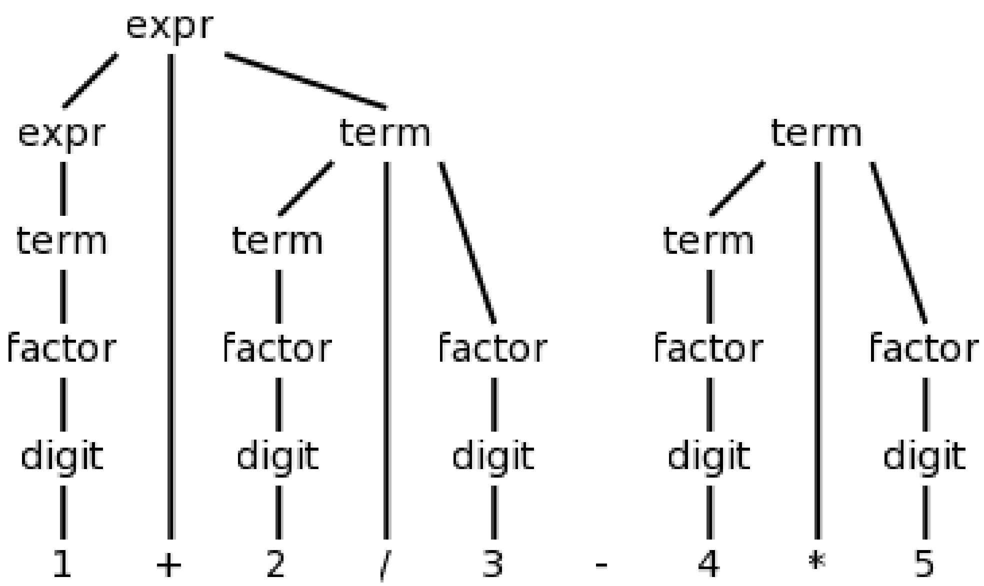
Syntax-Directed Translation

- Step8:



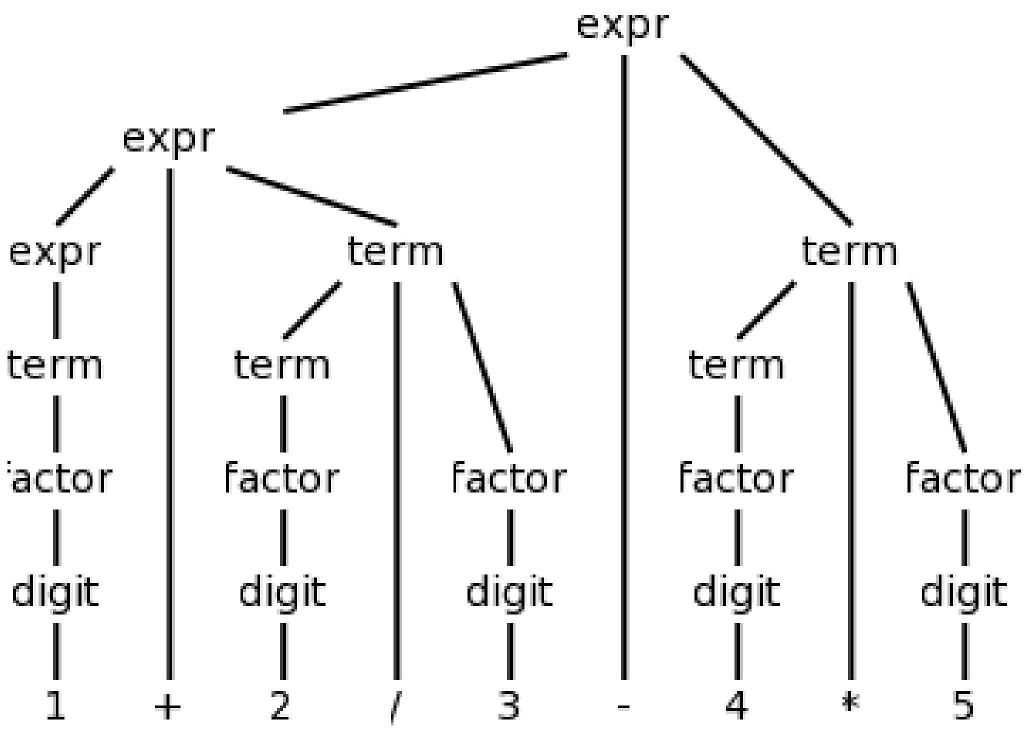
Syntax-Directed Translation

- Step9:



Syntax-Directed Translation

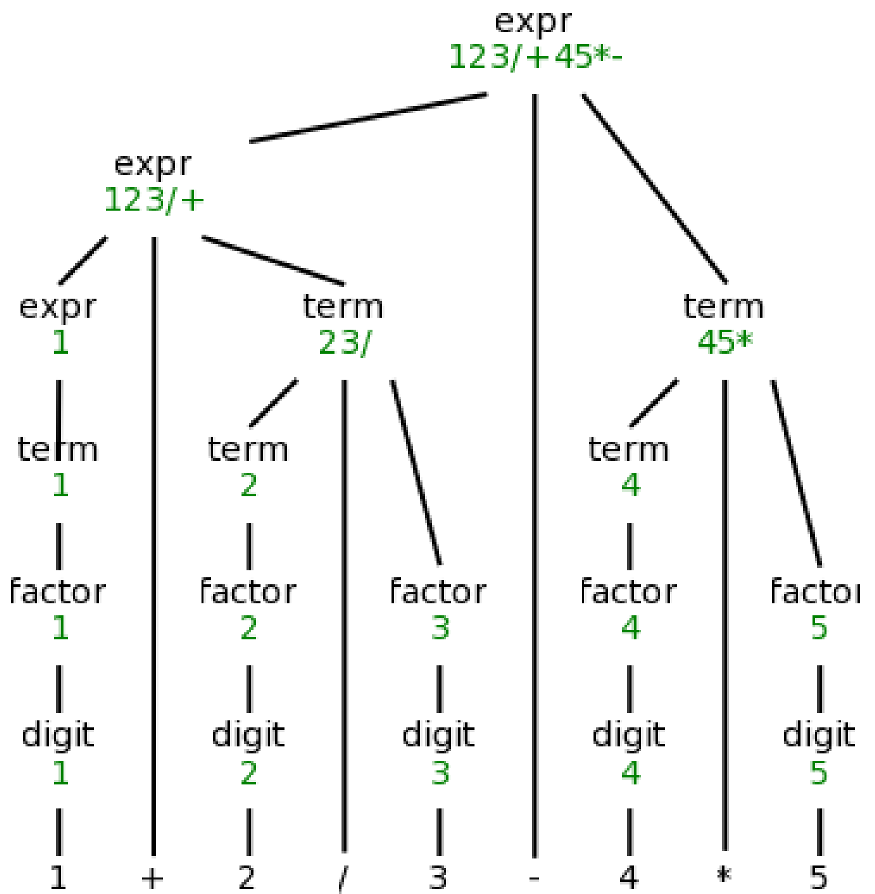
- Step10:



Syntax-Directed Translation

- Step11: (The last step, having all the internal nodes to associate the attributes)

```
expr   → expr + term | expr - term | term
term   → term * factor | term / factor | factor
factor → digit | ( expr )
digit  → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```



Syntax-Directed Translation

- Syntax-Directed Definitions (SDDs)
  - The attribute values at a node can depend on the values of attributes at the children of the node but not on attributes at the parent of the node.
  - We call such bottom-up attributes synthesized, since they are formed by synthesizing the attributes of the children.

Syntax-Directed Translation

- Simple Syntax-Directed Definitions
  - Book Chapter 2.3.3
  - TBD