CS419 Compilers Construction

A Simple One-Pass Compiler [Chapter 2]

Lecture 6

Edited by Dr. Ismail Hababeh

Adapted from slides by Dr. Mohammad Daoud

German Jordanian University

Originally from slides by Dr. Robert A. Engelen

After generating the tokens from the lexial analyser, this stage costruct the parsing tree for the complier to understand the operations

Parsing (Semantic Analyzer)

- **Syntax-directed definition SDD** builds up a translation by attaching attributes to the nodes of the parse-tree.

Extra infomation

- **Syntax-directed translation*** **(SDT)** is a method of translating a string into a sequence of actions by attaching one action to each rule of a grammar.

operation

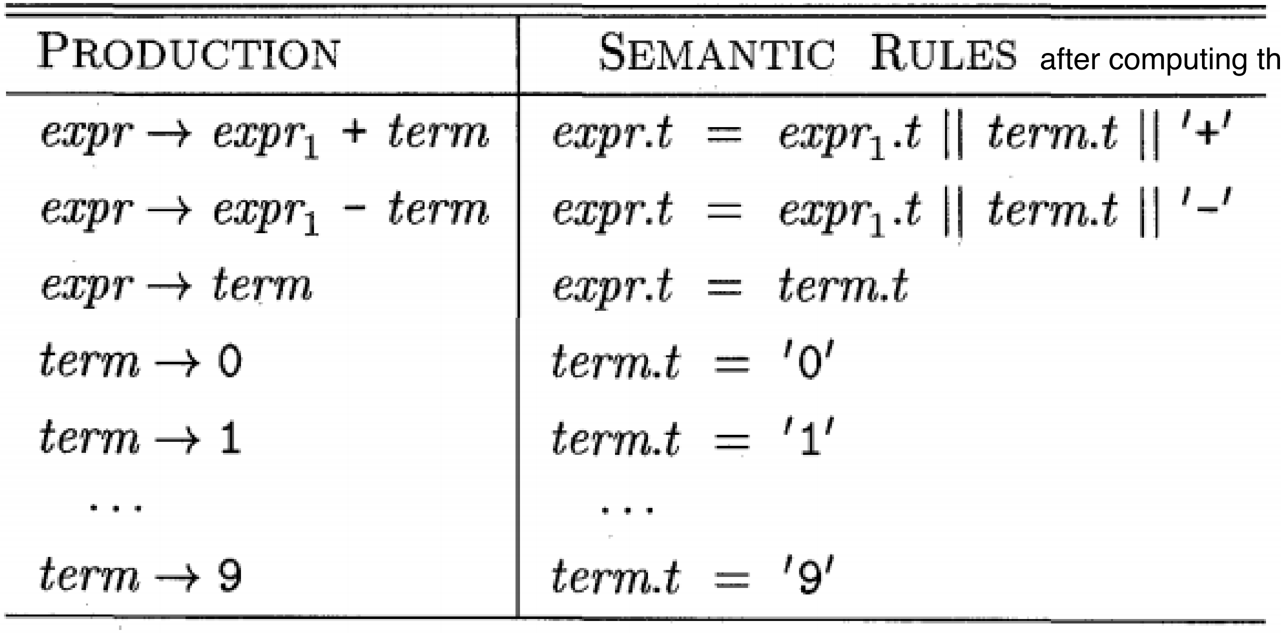
***Wikipedia**

Syntax-Directed Translation

- Associates a set of *attributes* (t) with terminals and non-terminals.
- Associates a set of *semantic rules* with each production to compute attributes’ values.
- The attributes contain the translated form of the input after computations are completed.

Attribute Grammar - Example

- Use the Syntax-directed definition SDD to translate expressions consisting of digits separated by + or – into postfix notation. So that the compiler can understand.
- The attribute t is associated with the non-terminals *expr1* and *term*.
- *expr1.t* denotes the attribute t value of expr1.
- The symbol || is the string concatenation operator.

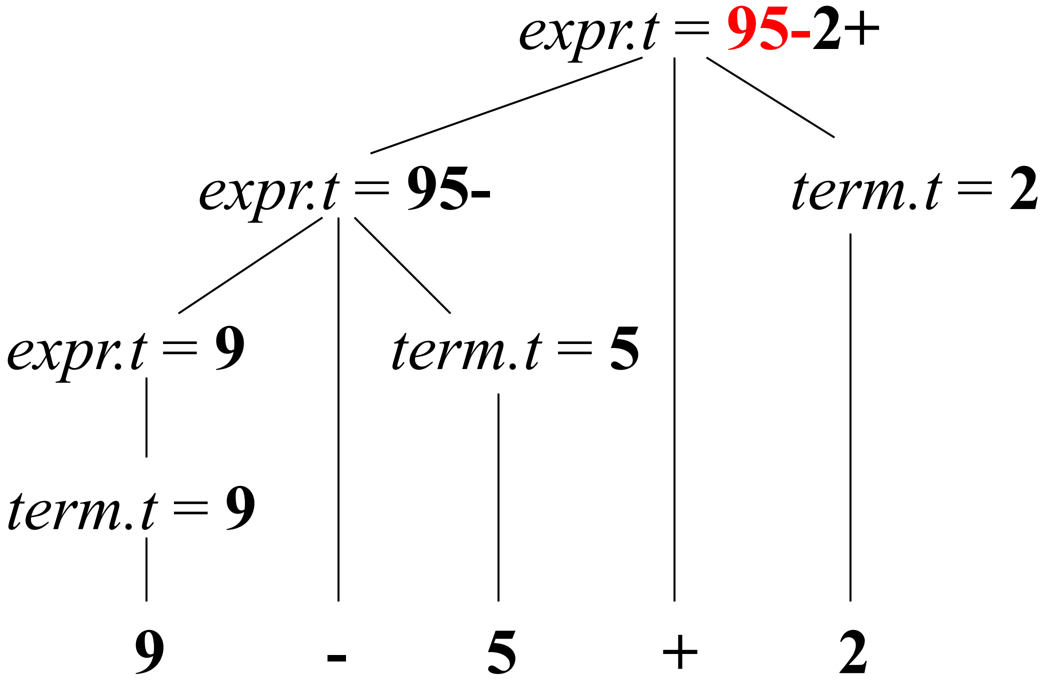


after computing the post fix notation

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

Annotated Parse Tree - Example

- **Annotated parse tree:** A parse tree showing the attribute values at each node.
- Example: the annotated parse tree of the expression 9-5+2 is described as follows



Synthesized and Inherited Attributes

الnode القبل اخيره

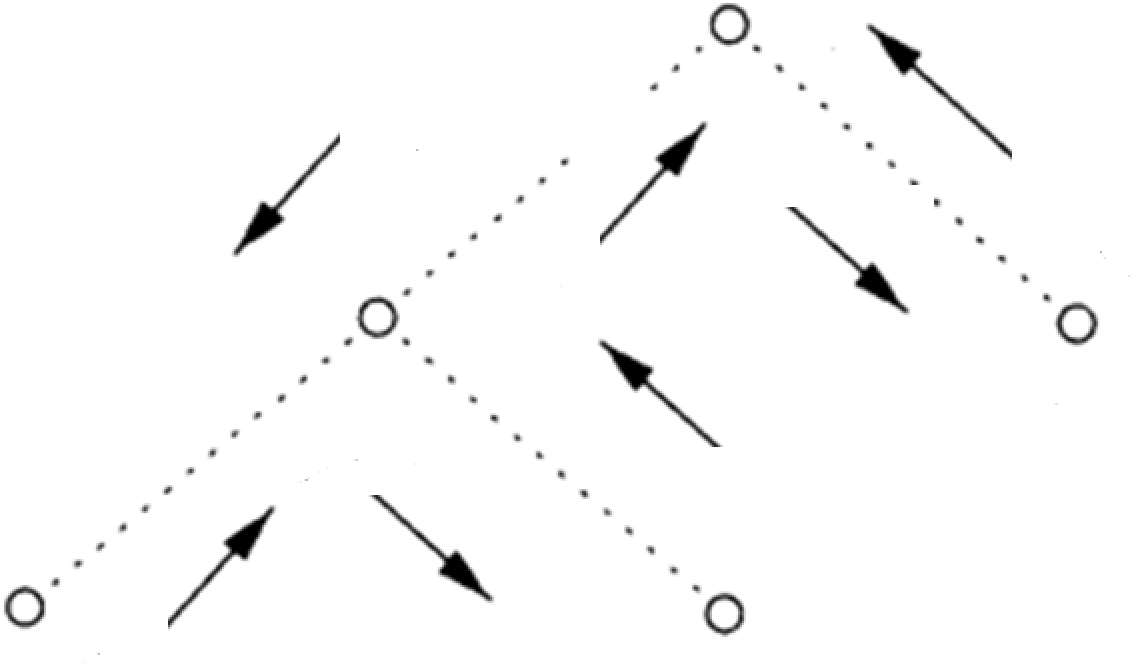
- The *Attribute is synthesized* if its value at a parse-tree node is determined from the attribute values at the children's nodes that can be evaluated during single bottom-up transversal of a parse tree.

- *The Attribute is inherited* if its value at a parse-tree node is determined by the parent (enforcing the parent’s semantic rules). This will be discussed later

Tree Transversal

- A syntax-directed definition does not require any specific order of attributes evaluation on a parse tree.
- Synthesized attributes can be evaluated using any *bottom-up* transversal

Depth-First Traversal Parse-Tree Route

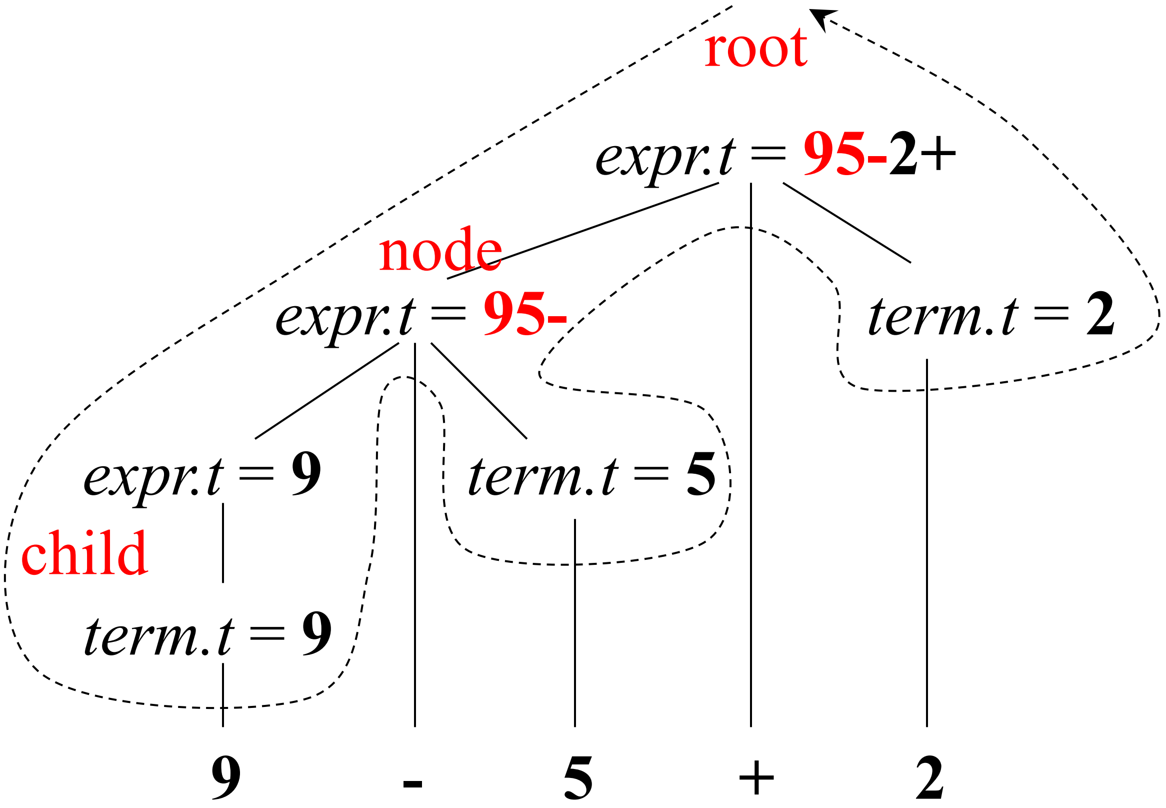


=> Starts from the root and recursively visits the children of each node in any order (here from left to right).

Depth-First Traversal Parse-Tree Pseudocode

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

Depth-First Traversal Parse-Tree- Example



Note: all attributes in this example are of the synthesized type.

ما هو اثر الDFS

لازم نعرف شغلتين، اولا احنا بدنا postfix notation و

لازم نعرف انو العمليه الرياضيه تابعه للnode يلي عندها terms

فعندما نعملDPS فاحنا بناخد الterms اول بعدين مناخد الoperation بس نخلص.

Translation Schemes

- A *translation scheme* is a CF grammar embedded with *semantic actions*

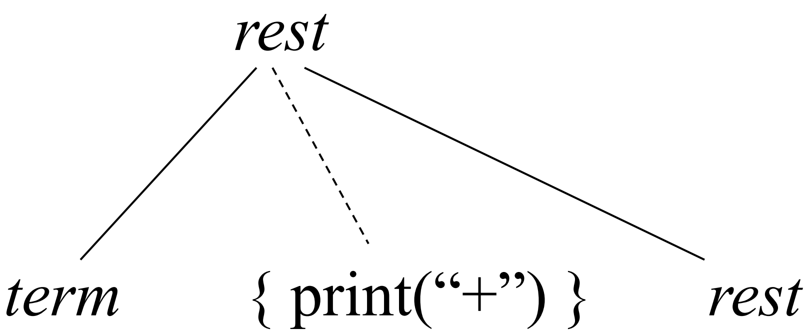
- An alternative way of translation is to use syntax-directed translation scheme that incrementally *attaches program fragments to productions* in a grammar

Attach program fragments to productions - Example

$$\mathit{rest} \rightarrow \mathit{term}\ \underbrace{\{\ \mathrm{print}(“+”)\ \}}_{\text{Embedded semantic action}}\ \mathit{rest}$$

Print is performed without the need to store attributes.

Parse-Tree



Parsing Methods

- Two methods of parsing:
- *Top-down parsing*: "constructs" a parse tree from root to leaves
    - *Recursive Descent Parsing*
    - *Predictive Parsing*
- *Bottom-up parsing*: "constructs" a parse tree from leaves to root

Recursive Descent Parsing

- *Recursive descent parsing* is a top-down parsing method:
  - Every non-terminal has one (recursive) procedure responsible for parsing the non-terminal's syntactic category of input tokens
  - When a non-terminal has multiple productions, each production is implemented in a branch of a selection (if … then …else) statement based on input look-ahead information.

Predictive Parsing

- *Predictive parsing* is a special form of recursive descent parsing where we use one lookahead (current) token to determine the parse operations.

Predictive Parser Grammar – Example

type → *simple*
| **^ id**
| **array [** *simple* **] of** *type*
simple → **integer**
| **char**
| **num dot num**

Predictive Parser - Program Code Example

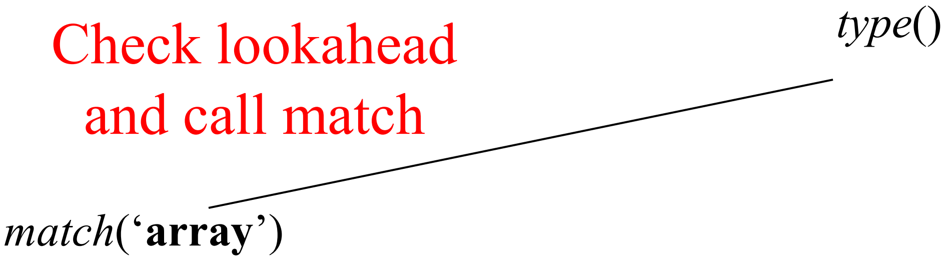
```
procedure type();
begin
    if lookahead in { ‘integer’, ‘char’, ‘num’ } then
        simple()
    else if lookahead = ‘^’ then
        match(‘^’); match(id)
    else if lookahead = ‘array’ then
        match(‘array’); match(‘[‘); simple();
        match(‘]’); match(‘of’); type()
    else error()
end;
```

```
type → simple
        | ^ id
        | array [ simple ] of type
simple → integer
        | char
        | num dot num
```

```
procedure simple();
begin
    if lookahead = ‘integer’ then
        match(‘integer’)
    else if lookahead = ‘char’ then
        match(‘char’)
    else if lookahead = ‘num’ then
        match(‘num’);
        match(‘dot’);
        match(‘num’)
    else error()
end;
```

```
procedure match(t : token);
begin
    if lookahead = t then
        lookahead := nexttoken()
    else error()
end;
```

Predictive Parser Example - Step 1

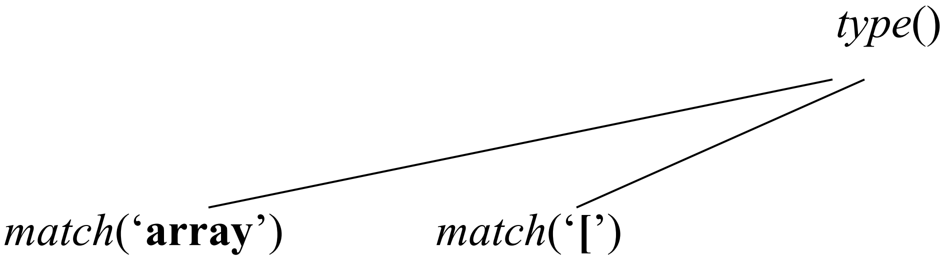


Check lookahead
and call match

<u>Input</u>: **array** **[** **num** **dot** **num** **]** **of** **integer**

↑
lookahead

Predictive Parser Example – Step 2



type()

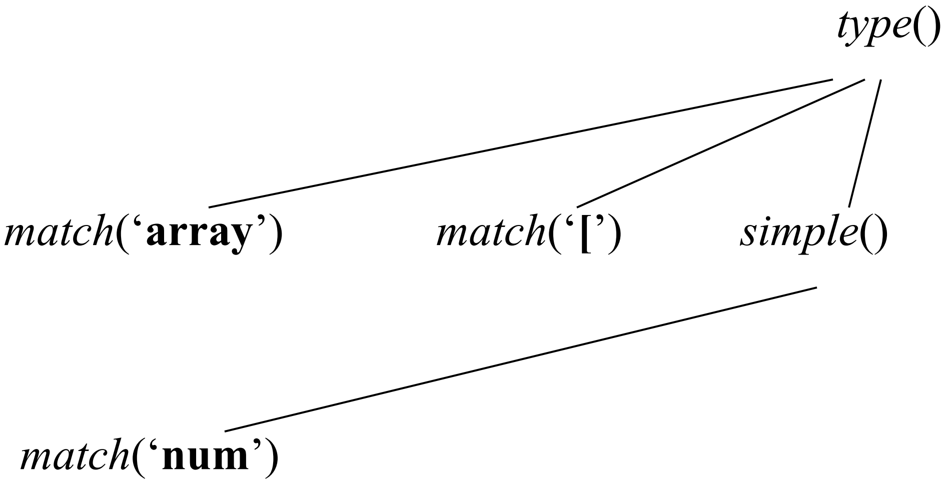
match(‘**array**’) *match*(‘**[**’)

Input: **array** **[** **num** **dot** **num** **]** **of** **integer**

↑

lookahead

Predictive Parser Example – Step 3



type()

match(‘**array**’) *match*(‘[’) *simple*()

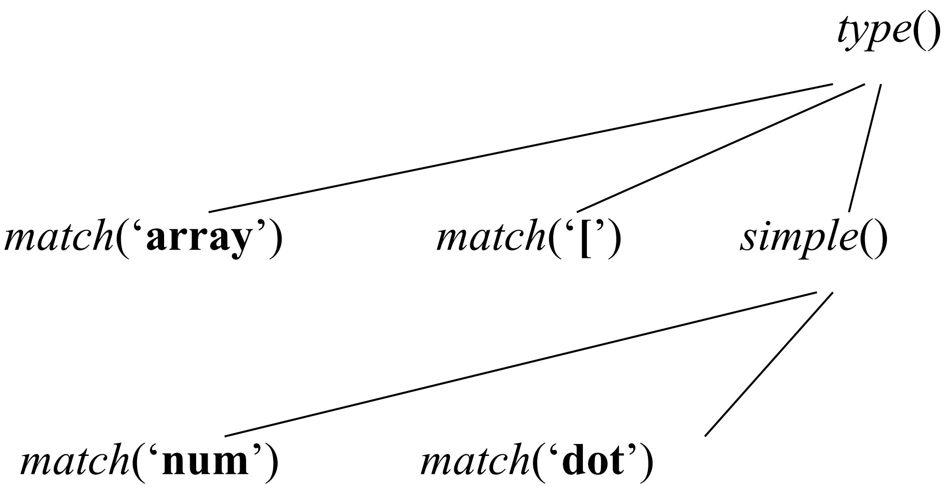
match(‘**num**’)

Input: **array** **[** **num** **dot** **num** **]** **of** **integer**

↑

lookahead

Predictive Parser Example – Step 4



type()

match(‘**array**’) *match*(‘**[**’) *simple*()

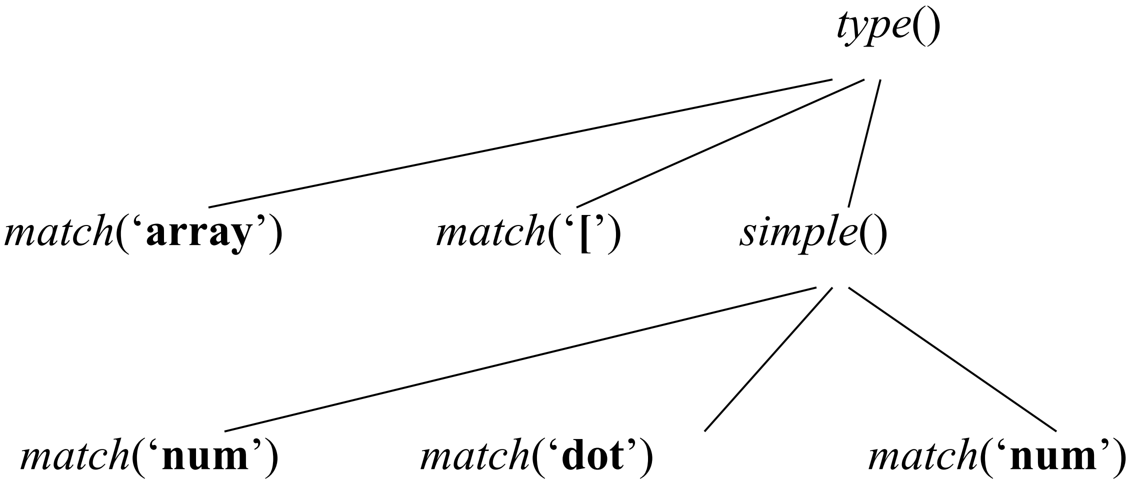
match(‘**num**’) *match*(‘**dot**’)

Input: **array** **[** **num** **dot** **num** **]** **of** **integer**

↑

lookahead

Predictive Parser Example – Step 5



type()

match(‘**array**’) *match*(‘**[**’) *simple*()

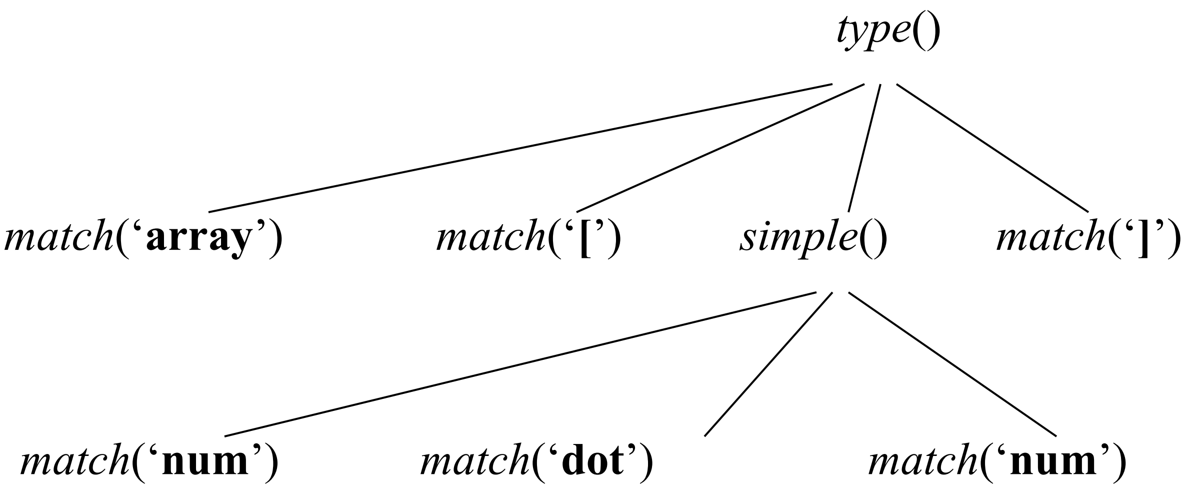
match(‘**num**’) *match*(‘**dot**’) *match*(‘**num**’)

Input: **array** **[** **num** **dot** **num** **]** **of** **integer**

↑

lookahead

Predictive Parser Example – Step 6



type()

match(‘**array**’) *match*(‘**[**’) *simple*() *match*(‘**]**’)

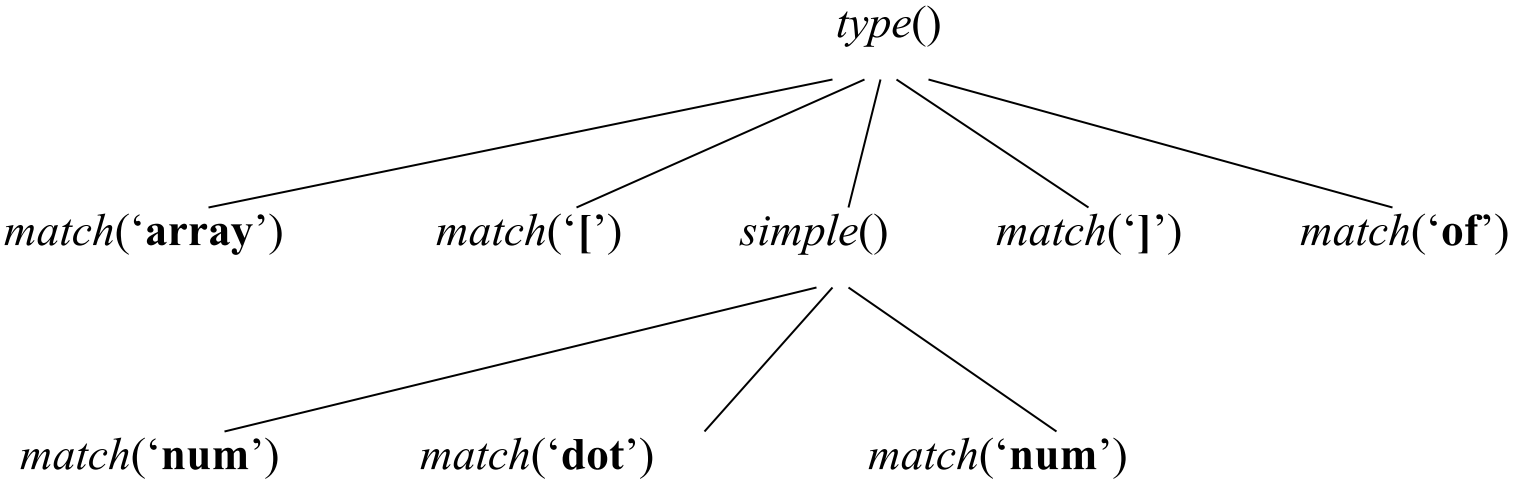
match(‘**num**’) *match*(‘**dot**’) *match*(‘**num**’)

Input: **array** **[** **num** **dot** **num** **]** **of** **integer**

↑

lookahead

Predictive Parser Example – Step 7



type()

match(‘**array**’) *match*(‘**[**’) *simple*() *match*(‘**]**’) *match*(‘**of**’)

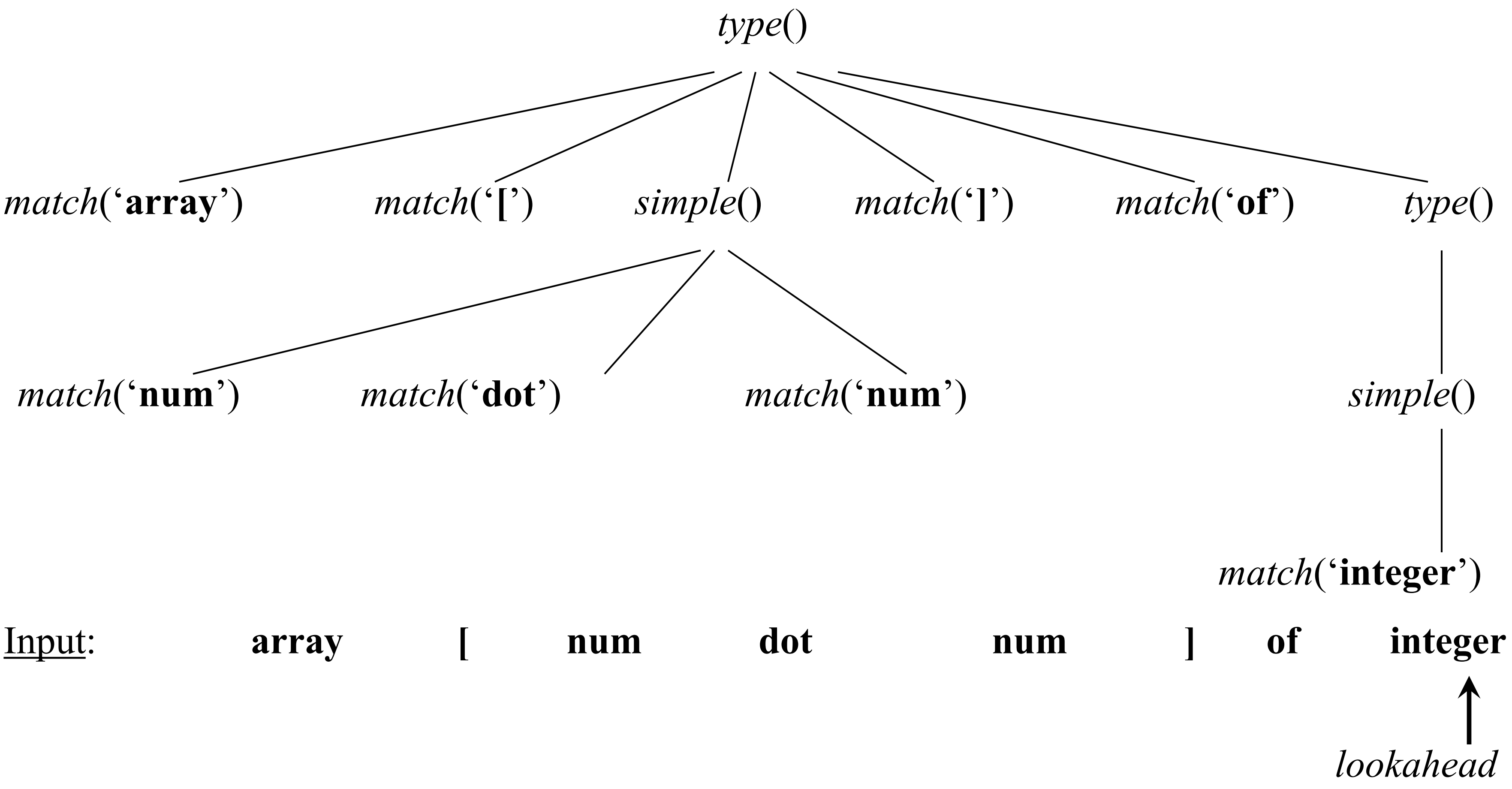
match(‘**num**’) *match*(‘**dot**’) *match*(‘**num**’)

Input: **array [num dot num] of integer**

↑

lookahead

Predictive Parser Example – Step 8



type()

match(‘**array**’) *match*(‘**[**’) *simple*() *match*(‘**]**’) *match*(‘**of**’) *type*()

match(‘**num**’) *match*(‘**dot**’) *match*(‘**num**’) *simple*()

match(‘**integer**’)

<u>Input</u>: **array** **[** **num** **dot** **num** **]** **of** **integer**

lookahead