CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

- Chapter 1 touches on all the material.
- Chapter 2 constructs (the front end of) a simple compiler.
- Chapters 3-8 fill in the (considerable) gaps, as well as the presenting the beginnings of the compiler back end

Syntax-Directed Translation

- Syntax-Directed Definitions (SDDs)
  - In our previous example,

  we give the **syntax directed definition** as follows:
  - '||' means concatenate

Production	Semantic Rule
expr → $expr_1$ + term	expr.t := $expr_1$.t || term.t || '+'
expr → $expr_1$ - term	expr.t := $expr_1$.t || term.t || '-'
expr → term	expr.t := term.t
term → $term_1$ * factor	term.t := $term_1$.t || factor.t || '*'
term → $term_1$ / factor	term.t := $term_1$.t || factor.t || '/'
term → factor	term.t := factor.t
factor → digit	factor.t := digit.t
factor → (expr)	factor.t := expr.t
digit → 0	digit.t := '0'
digit → 1	digit.t := '1'
digit → 2	digit.t := '2'
digit → 3	digit.t := '3'
digit → 4	digit.t := '4'
digit → 5	digit.t := '5'
digit → 6	digit.t := '6'
digit → 7	digit.t := '7'
digit → 8	digit.t := '8'
digit → 9	digit.t := '9'

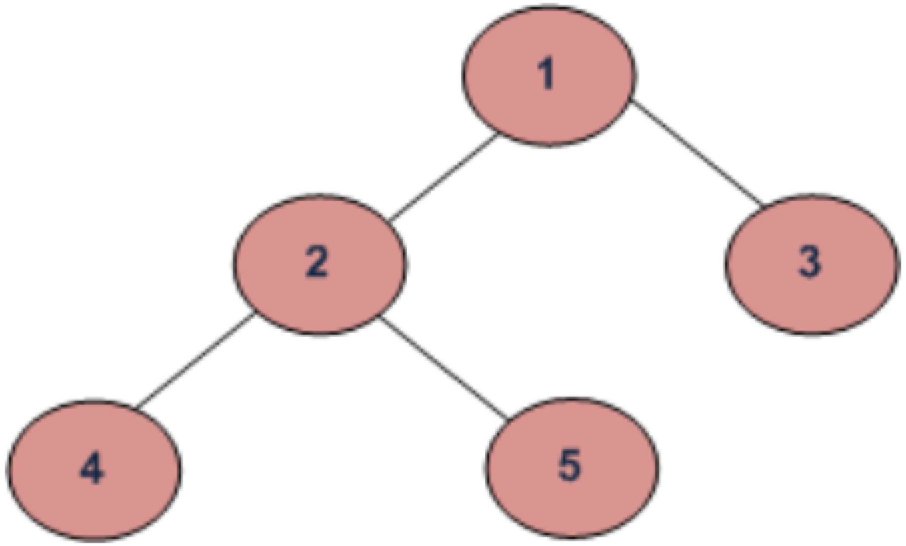
SDD for Infix to Posfix Translator

Syntax-Directed Translation

- Simple Syntax-Directed Definitions
  - Simple? Why it is called simple?
    - In the production (LHS) or in the semantic rule(RHS), if the left hand side (LHS) of the production is just the concatenation of the annotations for the non-terminals on the RHS **in the same order as the non-terminals appear in the production**, we call it simple
  - See the table of 1st one row
    - expr, expr1 and term, no matter it is in LHS RHS. If they are showing up in the same order (no need to care about the '+' ← terminal), it is called simple.

Syntax-Directed Translation

- **(depth-first) Tree Traversals (left to right)**
  - **The internal nodes (non-terminals, non-leaf) can be visited in on of the following approach**
    - Before visiting any of its children.
      - Pre-order, roots gets highest priority
    - Between visiting its children.
      - Left kid → root → right kid
    - After visiting all of its children.
      - Post-order, roots gets lowest priority



Depth First Traversals:

(a) Inorder (Left, Root, Right) : 4 2 5 1 3

(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth-First or Level Order Traversal: 1 2 3 4 5

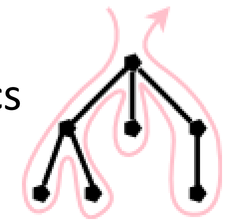
Syntax-Directed Translation

- **(depth-first) Tree Traversals (left to right) (Cont.)**
  - **I don’t like the pseudocode in book in the chapter 2.3.4**
  - **Very ambiguous, just 3 lines of code which confuses us a lot**
  - **I like this one more!**

```
traverse (n : treeNode)
    if leaf(n)                          -- visit leaves once; base of recursion
       visit(n)
    else                                -- interior node, at least 1 child
       -- visit(n)                      -- visit node PRE visiting any children
       traverse(first child)            -- recursive call
       while (more children remain)     -- excluding first child
           -- visit(n)                  -- visit node IN-between visiting children
           traverse (next child)        -- recursive call
       -- visit(n)                      -- visit node POST visiting all children
```

Syntax-Directed Translation

- **(depth-first) Tree Traversals (left to right) (Cont.)**
  - Note the following properties
    - Comments are introduced by -- and terminate at the end of the line
    - If you uncomment just the first (interior node) visit, you get a *preorder traversal*, in which each node is visited before (i.e., pre) visiting any of its children.
    - If you uncomment only the middle visit, you get an *inorder traversal*, in which the node is visited (in-) between visiting its children.
    - If you uncomment just the last visit, you get a *postorder traversal*, in which each node is visited after (i.e., post) visiting all of its children.
    - Inorder traversals are normally good for binary trees (exactly two children)
    - If you uncomment all of the three visits, you get an *Euler-tour traversal*.
      - An *Eulerian tour* on a directed graph is one that traverses each **edge** once.
      - If we view the tree on the right as undirected and replace each edge with two arcs each direction, we see that the pink curve is indeed an Eulerian tour.

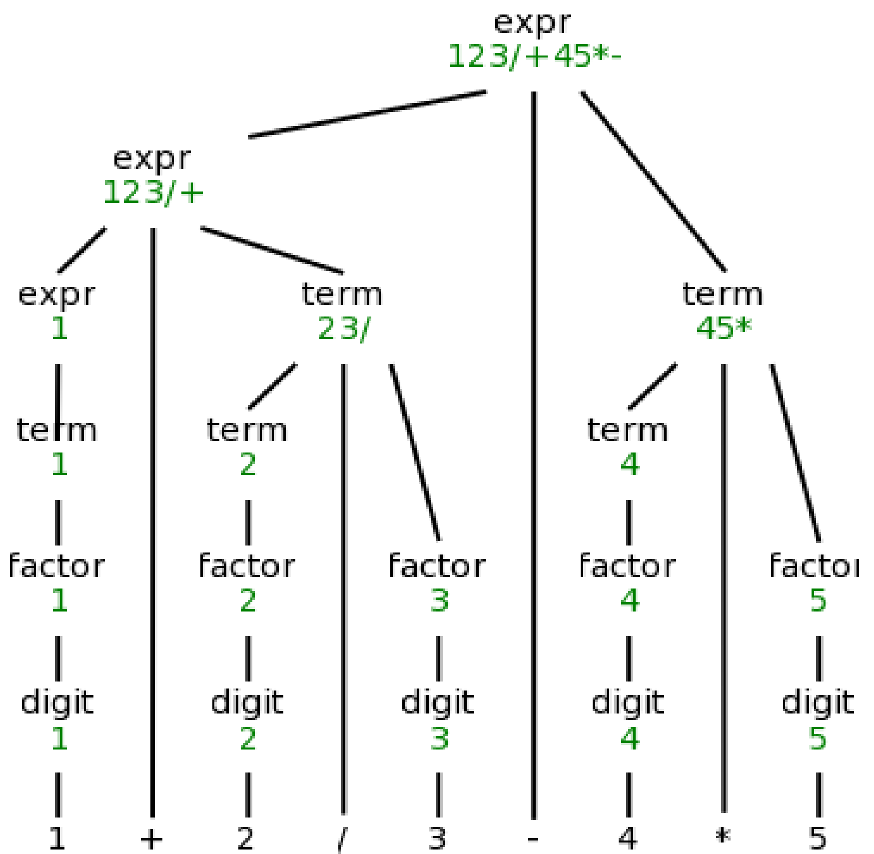


Syntax-Directed Translation

- **(depth-first) Tree Traversals (left to right) (Cont.)**
  - At this point in the course, we are considering only synthesized attributes, a postorder traversal will always yield a correct evaluation order for the attributes.
  - This is so since synthesized attributes depend only on attributes of child nodes and a postorder traversal visits a node only after all the children have been visited (and hence all the child node attributes have been evaluated).

Syntax-Directed Translation

- **Translation schemes**
  - The bottom-up annotation scheme just described generates the final result as the annotation of the root. (remember this?)
  - In our infix to postfix example,
    we get the result desired by
    printing the root annotation.
  - There is another technique,
    that produces its results incrementally.
  - There is a thing called **"semantic actions"**
  - When it is drawn in diagrams
  (e.g., see the diagram below),
  the semantic action is connected to its node,
  with a distinctive, often dotted, line.



Syntax-Directed Translation

- **Translation schemes (Cont.)**
  - **Definition**: A *syntax-directed translation scheme* is a context-free grammar with embedded semantic actions.
  - An example, in the infix to postfix translation translator, the parent either
    - takes the attribute of its only child, or
    - concatenates the attributes, left to right of its several children, and adds something at the end.
  - In practice, it is redundant to give both semantic actions and semantic rules. In this course we will emphasize semantic rules, i.e. syntax directed definitions (SDDs).
  - I show both the rules and the actions in a table just so that we can see the correspondence.

Syntax-Directed Translation

- **Translation schemes (Cont.)**
  - **Goal:** infix to postfix translation
  - The diagram for input string 1+2/3-4*5 with attached semantic actions is shown on the below.

Production with Semantic Action		Semantic Rule
expr → expr1 + term	{ print('+') }	expr.t := expr1.t || term.t || '+'
expr → expr1 - term	{ print('-') }	expr.t := expr1.t || term.t || '-'
term → term1 / factor	{ print('/') }	term.t := term1.t || factor.t || '/'
term → factor	{ null }	term.t := factor.t
digit → 3	{ print ('3') }	digit.t := '3'

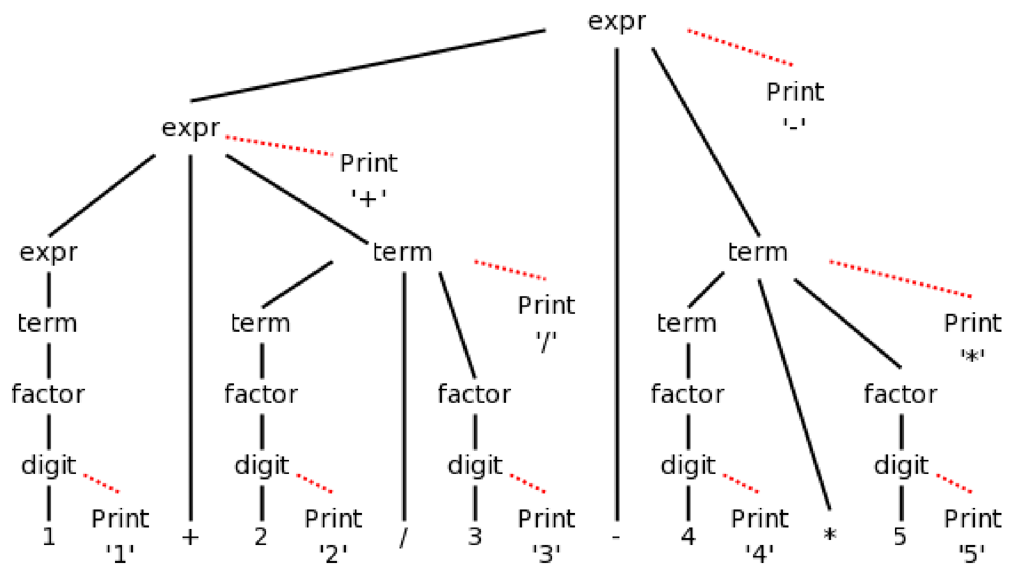
Semantic Actions and Rules for an Infix to Postfix Translator

  - We just do a (left-to-right) depth first traversal of the corresponding diagram and perform the semantic actions as they occur.
  - When these actions are print statements as above, we are said to be emitting the translation.

Syntax-Directed Translation

- **Translation schemes (Cont.)**
  - Since the actions are all leaves of the tree, they occur in the same order for any depth-first (left-to-right) traversal
  - Do on the board a depth first traversal of the diagram, performing the semantic actions as they occur, and confirm that the translation emitted is in fact 123/+45*-, the postfix version of 1+2/3-4*5
  - The way to traverse: left to right, all leaves!

(produced by red-dotted line)



Syntax-Directed Translation

- **Translation schemes (Cont.)**
  - **Goal:** prefix to infix translation
  - In our previous example, we haven't talk about '(' and ')'
    - Precedence
    - In this example, we want to demonstrate this capability
    - Think about this grammar, which generates the simple language of prefix expressions consisting of addition and subtraction of digits between 1 and 3 without parentheses
    - P → + P P | - P P | 1 | 2 | 3
    - Input string: +1-32
    - P→+PP → +P-PP→+P-32→+1-32
    - We can do something in the translation table to add the function to support '(' and ')'

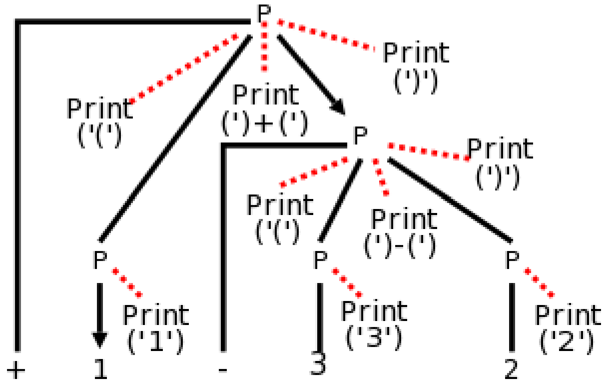
Syntax-Directed Translation

See if you can get the answer (1)+((3)-(2)) ???

- **Translation schemes (Cont.)**
  - **Here is the table (we are now working on the "action")**
  - **And here is the tree by applying the red-dotted actions**

Production with Semantic Action	Semantic Rule
$P \rightarrow +$ { print('(') } P_1 { print(')+(') } P_2 { print(')') }	P.t := '(' || P_1.t || ')+(' || P.t || ')'
$P \rightarrow -$ { print('(') } P_1 { print(')-(') } P_2 { print(')') }	P.t := '(' || P_1.t || ')-(' || P.t || ')'
$P \rightarrow 1$ { print('1') }	P.t := '1'
$P \rightarrow 2$ { print('2') }	P.t := '2'
$P \rightarrow 3$ { print('3') }	P.t := '3'

Prefix to infix translator



- **The way to traverse is the same! Depth-first,** left to right, all leaves!

Parsing

- In this section we assume that the lexical analyzer has already scanned the source input and converted it into a sequence of tokens.
- **Objective**: Given a string of tokens and a grammar, produce a parse tree yielding that string (or at least determine if such a tree exists).
- We will learn both top-down (begin with the start symbol, i.e. the root of the tree) and bottom up (begin with the leaves) technique
- In this chapter we just quickly go through bottom-up quickly.
- And little bit more in doing top-down, which is easier. In Ch.4 we will cover both (in detail).

Parsing

- **Bottom-up parsing**
  - **Input string: a + b * c (forget about the precedence)**
  - **The goal is try to reduce (not derive) the input string to the root**
  - **Production rules:**

```
S → E
E → E + T
E → E * T
E → T
T → id
```

  - Read the input and check if any production matches with the input:
    - It can be reduced into S !
    - Try to build up your (bottom-up) parse tree!

```
a + b * c
T + b * c
E + b * c
E + T * c
E * c
E * T
E
S
```

Parsing

- Top-down parsing
  - Put the root on top is your first step!
  - Top (root) to down (leaves)
  - Given the following Pascal-like simple language, a high level idea, not the real Pascal (way more complicated)
    - 2 non-terminals
      - type: start symbol
      - simple: means the "simple type"
    - 8 terminals
      - Tokens already produced by the lexer

1. **integer** and **char**
2. **id** for identifier
3. **array** and **of** used in array declarations
4. ↑ meaning pointer to
5. **num** for a (positive whole) number
6. **dotdot** for .. (used to give a range like 6..9)

Parsing

- Top-down parsing (Cont.)

The productions are

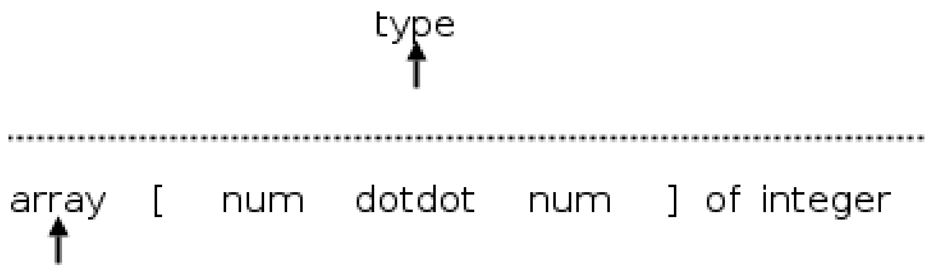
```
type   → simple
type   → ↑ id
type   → array [ simple ] of type
simple → integer
simple → char
simple → num dotdot num
```

- Here is a movie showing you,

the whole process in building the whole tree

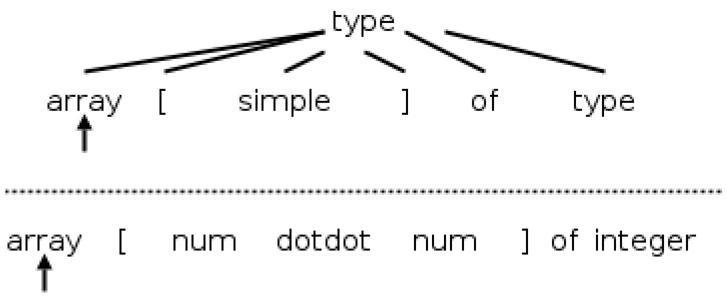
(Up: Tree to be built, Down: input string)

- Step1

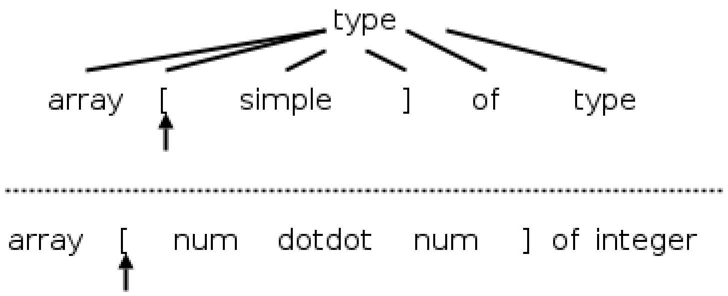


Parsing

- Top-down parsing (Cont.)
  - Step2

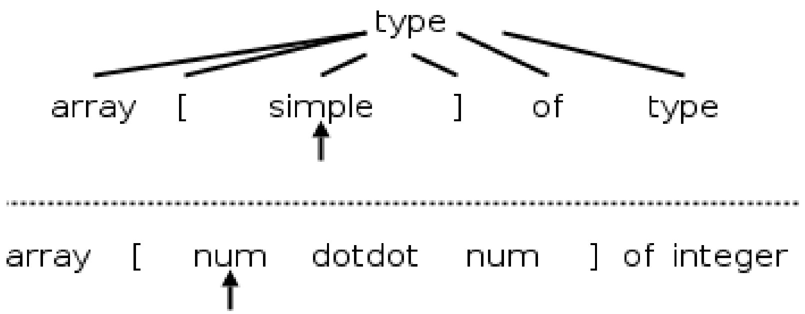


  - Step3

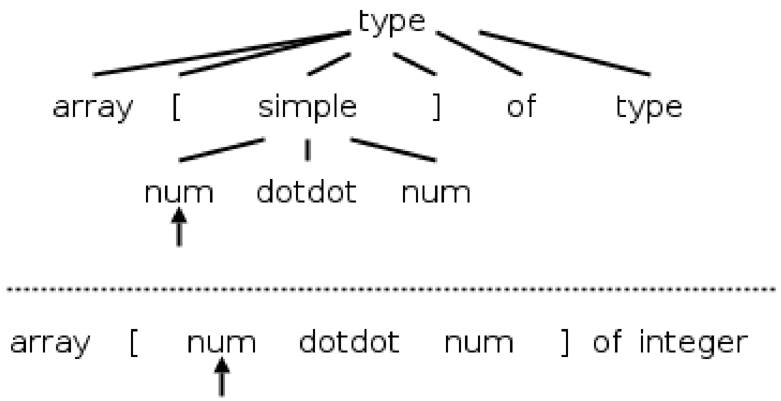


Parsing

- Top-down parsing (Cont.)
  - Step4

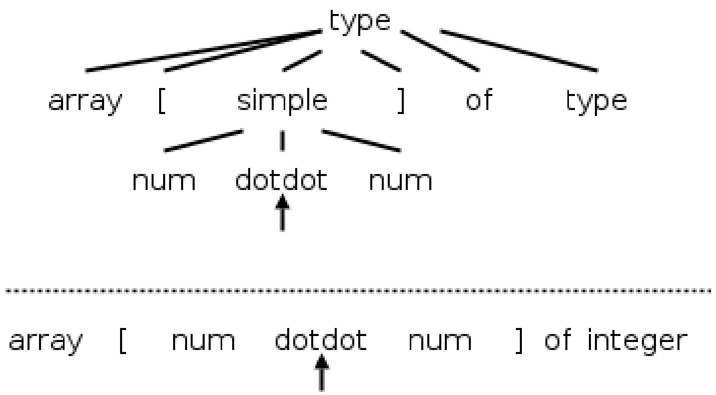


  - Step5

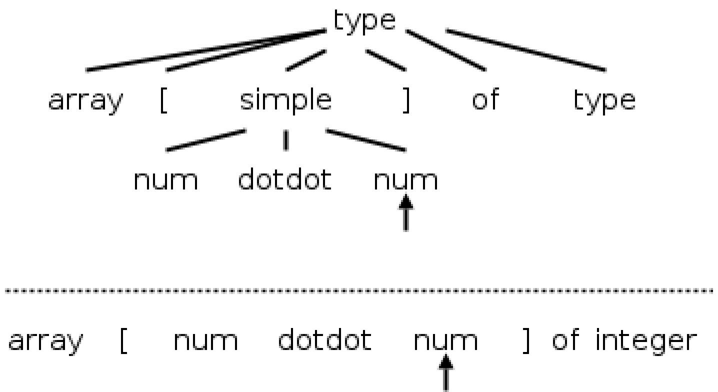


Parsing

- Top-down parsing (Cont.)
  - Step6

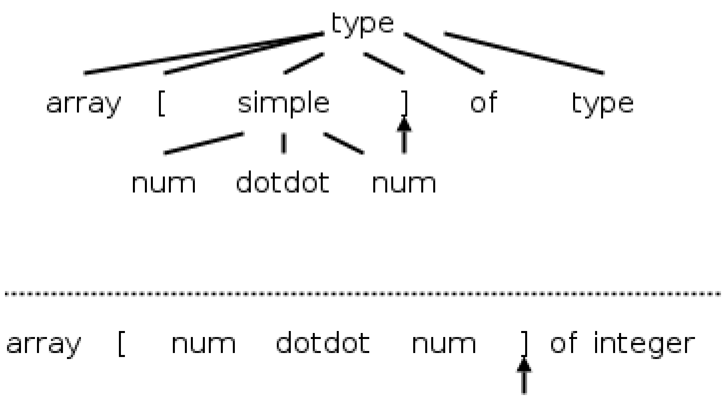


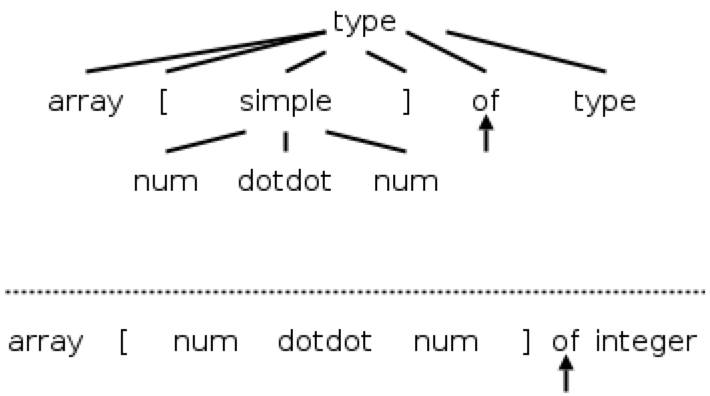
  - Step7

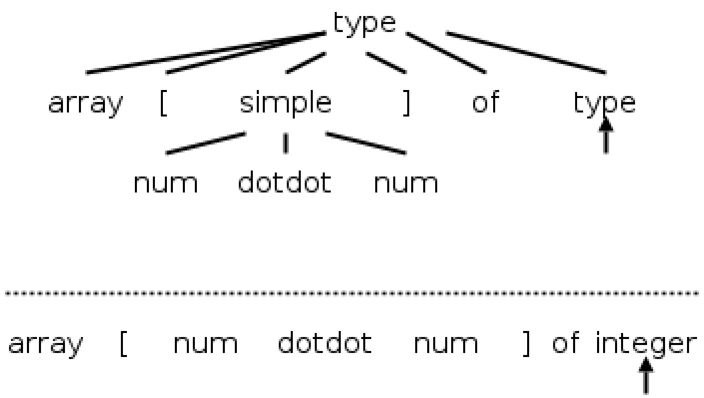


Parsing

- Top-down parsing (Cont.)
  - Step8~10

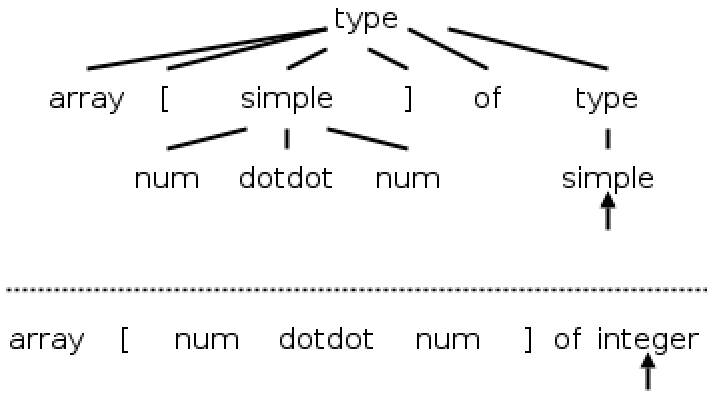


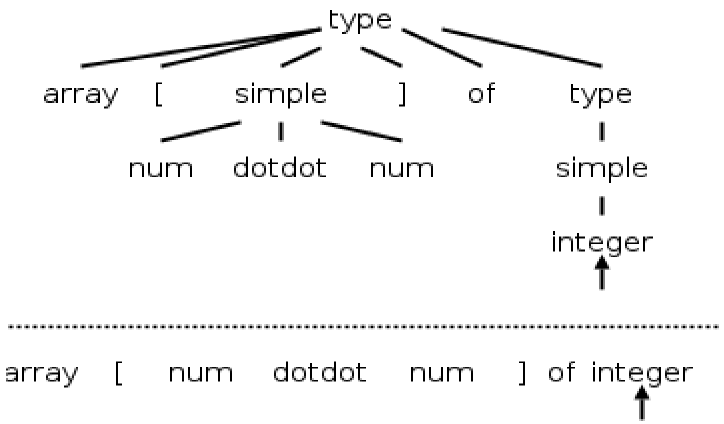


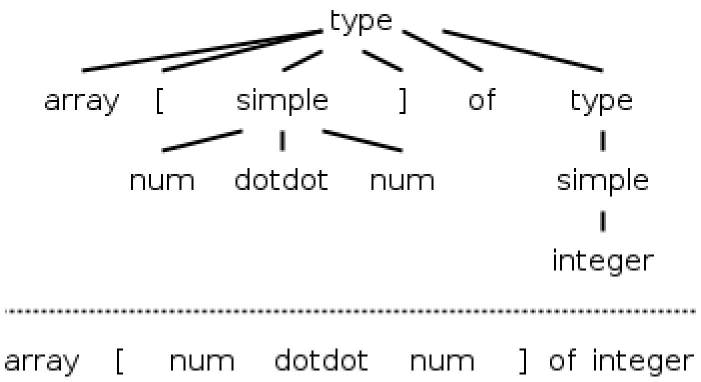


Parsing

- Top-down parsing (Cont.)
  - Step11~13







Parsing

- Predictive Parsing
  - TBD, in Part3