

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

Parsing (Semantic Analyzer)

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

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.

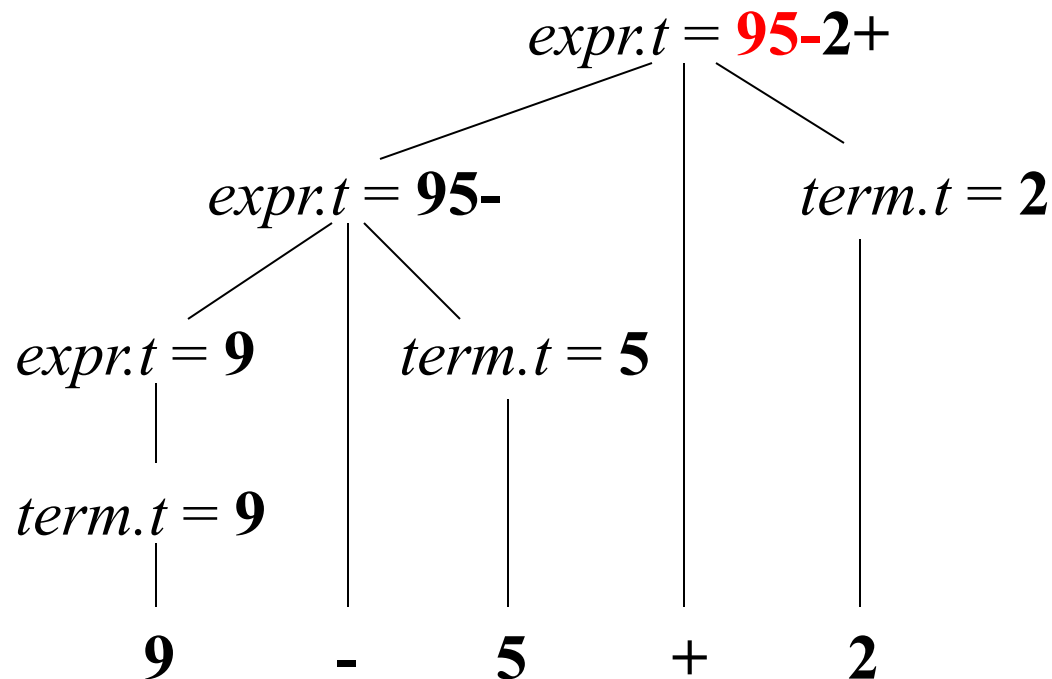
Attribute Grammar - Example

- Use the Syntax-directed definition SDD to **translate expressions** consisting of **digits** separated by **+** or **-** into **postfix** notation.
- 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**.

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



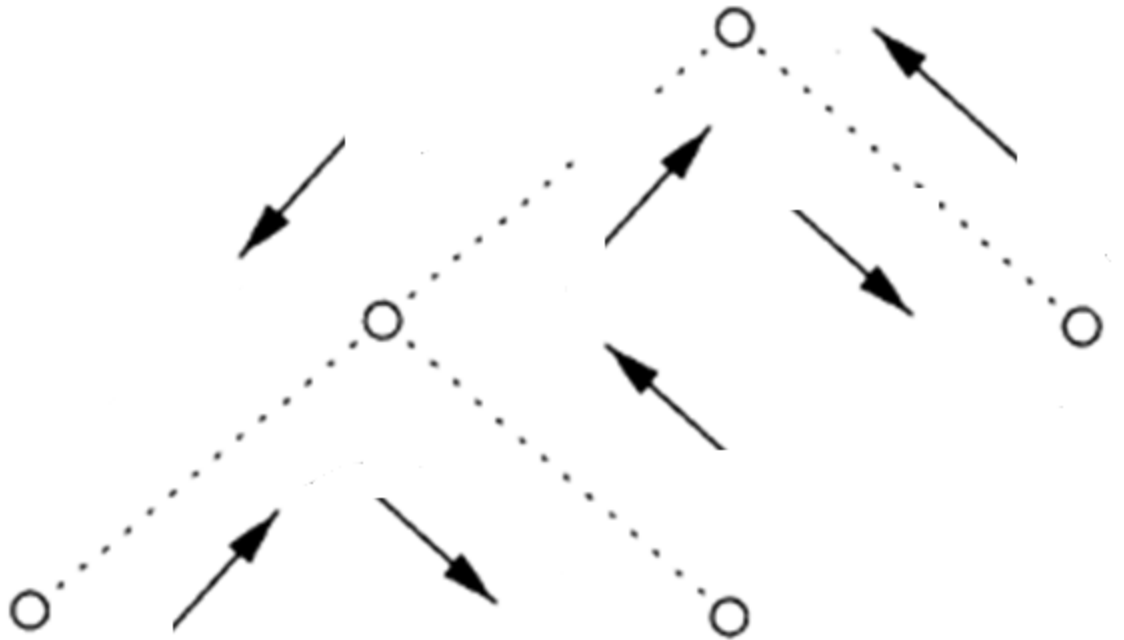
Synthesized and Inherited Attributes

- 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

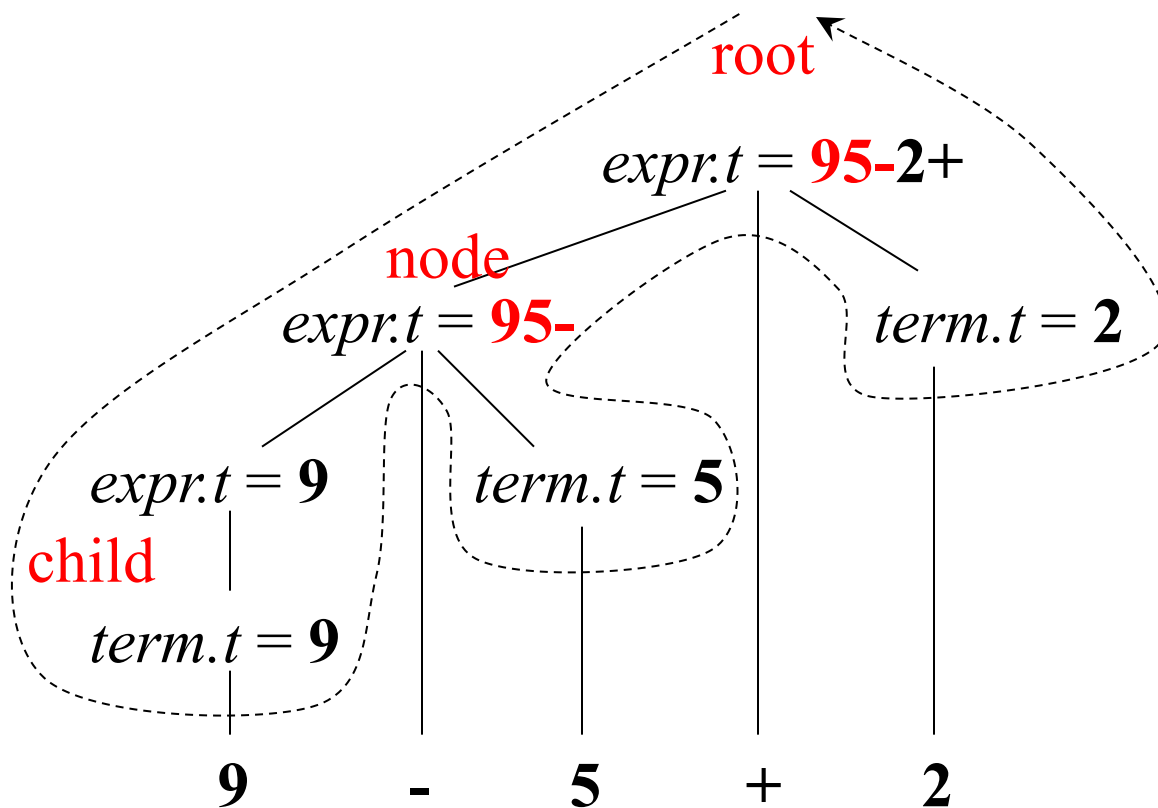


=> Starts from the root and recursively visits the children of each node in any order (here from left to right).

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.


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

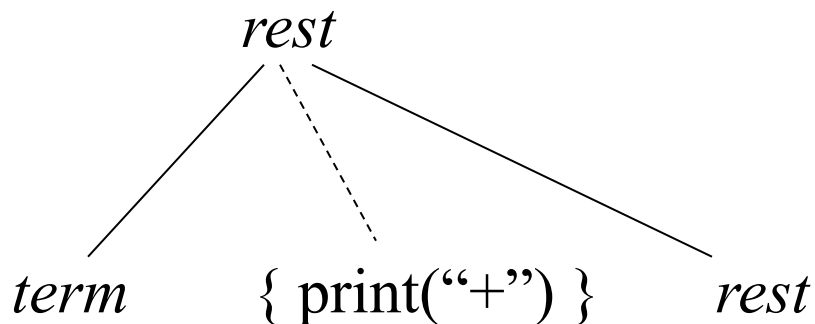
Attach program fragments to productions - Example

$rest \rightarrow term \{ \text{print}(\text{"+"}) \} rest$

Print is performed without the need to store attributes.


Embedded
semantic action

Parse-Tree



Parsing Methods

- Two methods of parsing:
- *Top-down parsing*: “constructs” a parse tree from root to leaves
 - *Recursive Descent Parsing*: it may encounter difficulties with left-recursive grammar rules since they can result in unbounded recursion. *backtracking is required in case of failure.*
 - *Predictive Parsing*: *no backtracking is required*
- *Bottom-up parsing*: “constructs” a parse tree from leaves to root

Top-Down 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.

Top-Down 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

- Write the predictive parser procedures for the following grammar:

type \rightarrow *simple*
 | **^ id**
 | **array [*simple*] of *type***
simple \rightarrow **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;

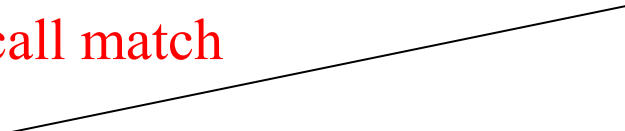
```

Predictive Parser Example - Step 1

Check lookahead
and call match

match('array')

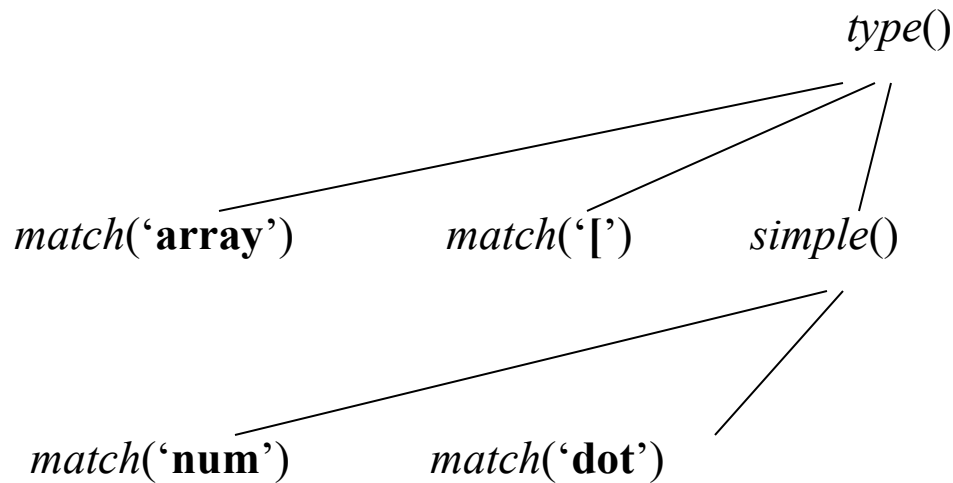
type()



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

 ↑
 lookahead

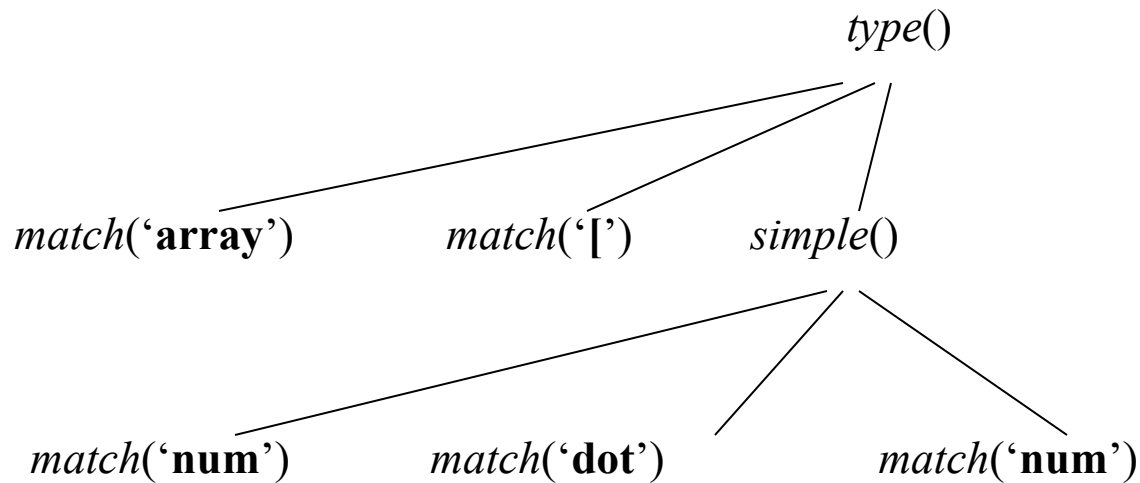
Predictive Parser Example – Step 4



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

↑
lookahead

Predictive Parser Example – Step 5



Input:

array [num dot num] of integer

↑
lookahead

