

COP4020

Programming

Languages

Syntax

Robert van Engelen & Chris Lacher



Overview

- Tokens and regular expressions
- Syntax and context-free grammars
- Grammar derivations
- More about parse trees
- Top-down and bottom-up parsing
- Recursive descent parsing

Tokens

- Tokens are the basic building blocks of a programming language
 - Keywords, identifiers, literal values, operators, punctuation
- We saw that the first compiler phase (scanning) splits up a character stream into tokens
- Tokens have a special role with respect to:
 - *Free-format languages*: source program is a sequence of tokens and horizontal/vertical position of a token on a page is unimportant (e.g. Pascal)
 - *Fixed-format languages*: indentation and/or position of a token on a page is significant (early Basic, Fortran, Haskell)
 - *Case-sensitive languages*: upper- and lowercase are distinct (C, C++, Java)
 - *Case-insensitive languages*: upper- and lowercase are identical (Ada, Fortran, Pascal)

Defining Token Patterns with Regular Expressions

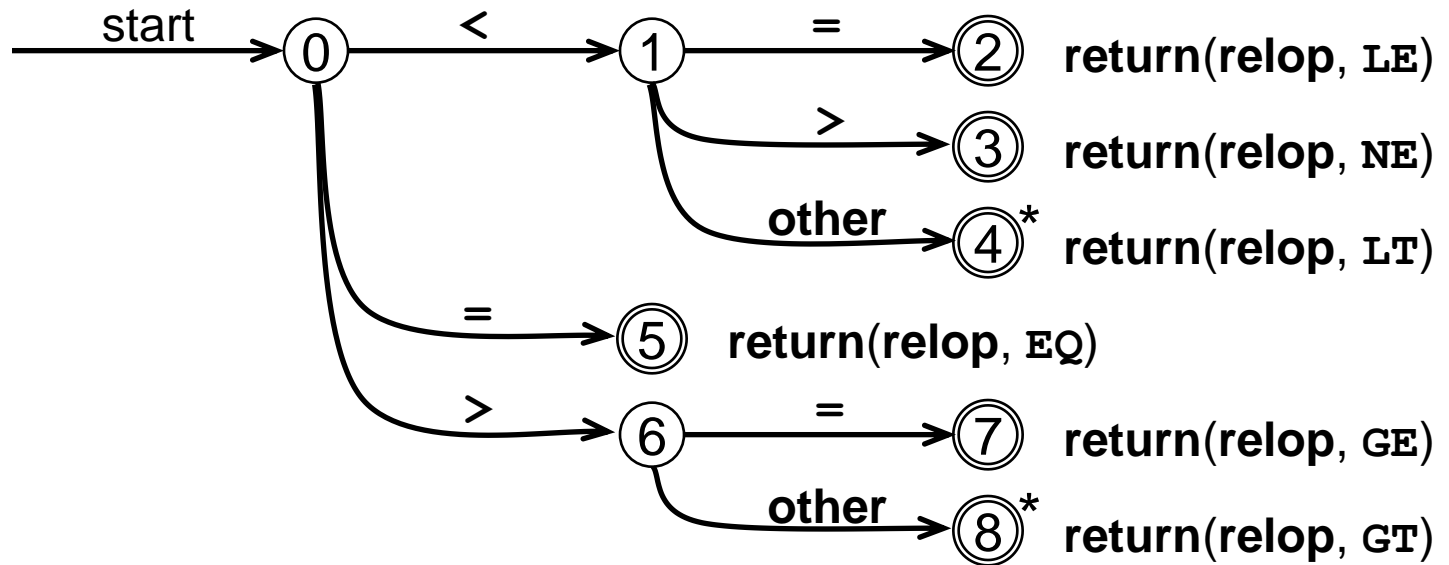
- The makeup of a token is described by a *regular expression (RE)*
- A regular expression r is one of
 - A character (an element of the RE alphabet), e.g.
 a
 - Empty, denoted by
 ε
 - *Concatenation*: a sequence of regular expressions
 $r_1 r_2 r_3 \dots r_n$
 - *Alternation*: regular expressions separated by a bar
 $r_1 | r_2$
 - *Repetition*: a regular expression followed by a star (Kleene star)
 r^*

Example Regular Definitions for Tokens

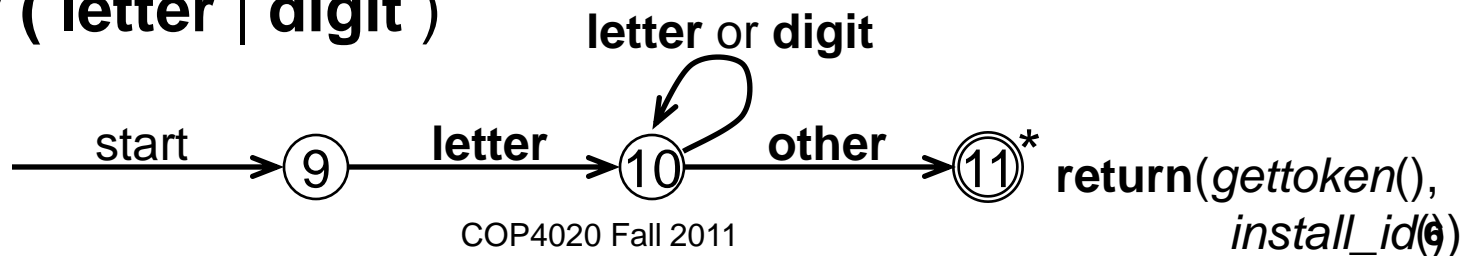
- $digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
- $unsigned_integer \rightarrow digit\ digit^*$
- $signed_integer \rightarrow (+ \mid - \mid \varepsilon)\ unsigned_integer$
- $letter \rightarrow a \mid b \mid \dots \mid z \mid A \mid B \mid \dots \mid Z$
- $identifier \rightarrow letter (letter \mid digit)^*$
- Cannot use recursive definitions! This is illegal:
 $digits \rightarrow digit\ digits \mid digit$

Finite State Machines = Regular Expression Recognizers

relop → < | <= | <> | > | >= | =



id → letter (letter | digit)*



Context Free Grammars: BNF

- Regular expressions cannot describe nested constructs, but *context-free grammars* can
- Backus-Naur Form (BNF) *grammar productions* are of the form

$\langle \text{nonterminal} \rangle ::= \text{sequence of (non)terminals}$

where

- A *terminal* of the grammar is a *token*
- A $\langle \text{nonterminal} \rangle$ defines a syntactic category
- The symbol $|$ denotes alternative forms in a production
- The special symbol ε denotes empty

Example

<Program> ::= **program** *<id>* (*<id>* *<More_ids>*); *<Block>* .
<Block> ::= *<Variables>* **begin** *<Stmt>* *<More_Stmts>* **end**
<More_ids> ::= , *<id>* *<More_ids>*
| ϵ
<Variables> ::= **var** *<id>* *<More_ids>* : *<Type>* ; *<More_Variables>*
| ϵ
<More_Variables> ::= *<id>* *<More_ids>* : *<Type>* ; *<More_Variables>*
| ϵ
<Stmt> ::= *<id>* := *<Exp>*
| **if** *<Exp>* **then** *<Stmt>* **else** *<Stmt>*
| **while** *<Exp>* **do** *<Stmt>*
| **begin** *<Stmt>* *<More_Stmts>* **end**
<More_Stmts> ::= ; *<Stmt>* *<More_Stmts>*
| ϵ
<Exp> ::= *<num>*
| *<id>*
| *<Exp>* + *<Exp>*
| *<Exp>* - *<Exp>*

Extended BNF

- *Extended* BNF adds
 - Optional constructs with [and]
 - Repetitions with []*
 - Some EBNF definitions also add []+ for non-zero repetitions

Example

<Program> ::= **program** *<id>* (*<id>* [, *<id>*]*); *<Block>* .

<Block> ::= [*<Variables>*] **begin** *<Stmt>* [; *<Stmt>*]* **end**

<Variables> ::= **var** [*<id>* [, *<id>*]* : *<Type>* ;]⁺

<Stmt> ::= *<id>* := *<Exp>*
| **if** *<Exp>* **then** *<Stmt>* **else** *<Stmt>*
| **while** *<Exp>* **do** *<Stmt>*
| **begin** *<Stmt>* [; *<Stmt>*]* **end**

<Exp> ::= *<num>*
| *<id>*
| *<Exp>* + *<Exp>*
| *<Exp>* - *<Exp>*

Derivations

- From a grammar we can *derive* strings by generating sequences of tokens directly from the grammar (the opposite of parsing)
- In each *derivation step* a nonterminal is replaced by a right-hand side of a production for that nonterminal
- The representation after each step is called a *sentential form*
- When the nonterminal on the far right (left) in a sentential form is replaced in each derivation step the derivation is called *right-most* (*left-most*)
- The final form consists of terminals only and is called the *yield* of the derivation
- A context-free grammar is a generator of a context-free language: the language defined by the grammar is the set of all strings that can be derived

Example

$\langle expression \rangle$::= identifier
	unsigned_integer
	- $\langle expression \rangle$
	($\langle expression \rangle$)
	$\langle expression \rangle$ $\langle operator \rangle$ $\langle expression \rangle$
$\langle operator \rangle$::= + - * /

$\langle expression \rangle$

$\Rightarrow \langle expression \rangle \langle operator \rangle$ $\langle expression \rangle$

$\Rightarrow \langle expression \rangle \langle operator \rangle$ identifier

\Rightarrow $\langle expression \rangle$ + identifier

$\Rightarrow \langle expression \rangle \langle operator \rangle$ $\langle expression \rangle$ + identifier

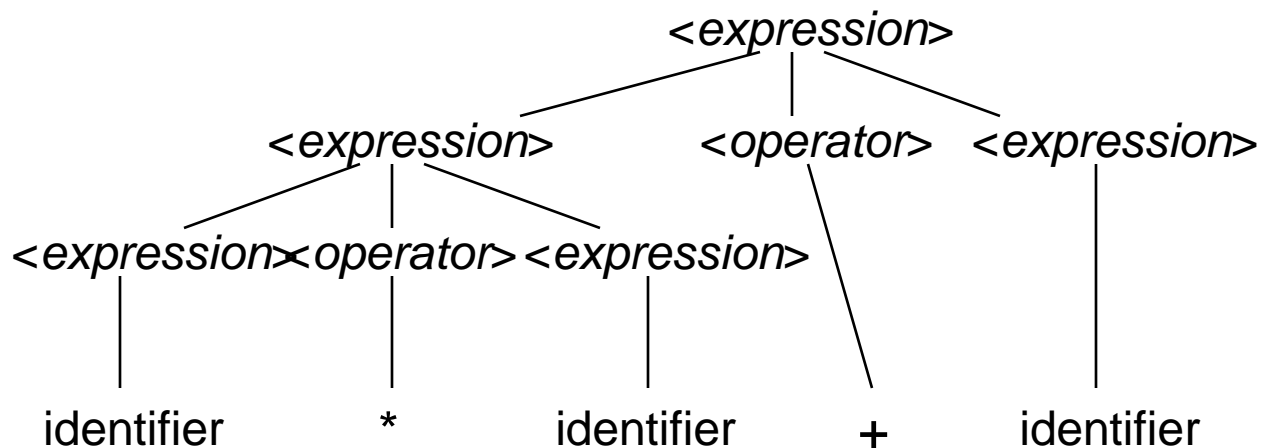
$\Rightarrow \langle expression \rangle \langle operator \rangle$ identifier + identifier

\Rightarrow $\langle expression \rangle$ * identifier + identifier

\Rightarrow identifier * identifier + identifier

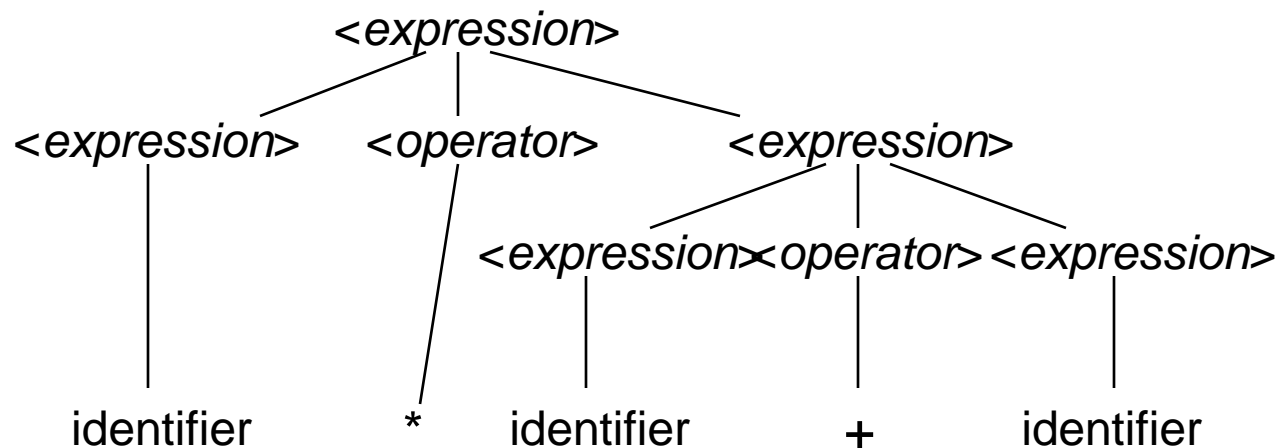
Parse Trees

- A *parse tree* depicts the end result of a derivation
 - The *internal nodes* are the nonterminals
 - The *children* of a node are the symbols (terminals and nonterminals) on a right-hand side of a production
 - The *leaves* are the terminals



Ambiguity

- There is another parse tree for the same grammar and input: the grammar is *ambiguous*
- This parse tree is not desired, since it appears that + has precedence over *



Ambiguous Grammars

- When more than one distinct derivation of a string exists resulting in distinct parse trees, the grammar is ambiguous
- A programming language construct should have only one parse tree to avoid misinterpretation by a compiler
- For expression grammars, associativity and precedence of operators is used to disambiguate the productions

$\langle expression \rangle$	$::= \langle term \rangle \mid \langle expression \rangle \langle add_op \rangle \langle term \rangle$
$\langle term \rangle$	$::= \langle factor \rangle \mid \langle term \rangle \langle mult_op \rangle \langle factor \rangle$
$\langle factor \rangle$	$::= \text{identifier} \mid \text{unsigned_integer} \mid - \langle factor \rangle \mid (\langle expression \rangle)$
$\langle add_op \rangle$	$::= + \mid -$
$\langle mult_op \rangle$	$::= * \mid /$

Ambiguous if-then-else

- A classical example of an ambiguous grammar are the grammar productions for if-then-else:

$$\begin{aligned} \langle \text{stmt} \rangle ::= & \mathbf{if} \langle \text{expr} \rangle \mathbf{then} \langle \text{stmt} \rangle \\ & | \mathbf{if} \langle \text{expr} \rangle \mathbf{then} \langle \text{stmt} \rangle \mathbf{else} \langle \text{stmt} \rangle \end{aligned}$$

- It is possible to hack this into unambiguous productions for the same syntax, but the fact that it is not easy indicates a problem in the programming language design
- Ada uses different syntax to avoid ambiguity:

$$\begin{aligned} \langle \text{stmt} \rangle ::= & \mathbf{if} \langle \text{expr} \rangle \mathbf{then} \langle \text{stmt} \rangle \mathbf{end} \mathbf{if} \\ & | \mathbf{if} \langle \text{expr} \rangle \mathbf{then} \langle \text{stmt} \rangle \mathbf{else} \langle \text{stmt} \rangle \mathbf{end} \mathbf{if} \end{aligned}$$

Linear-Time Top-Down and Bottom-Up Parsing

- A *parser* is a recognizer for a context-free language
- A string (token sequence) is accepted by the parser and a parse tree can be constructed if the string is in the language
- For any arbitrary context-free grammar parsing can take as much as $O(n^3)$ time, where n is the size of the input
- There are large classes of grammars for which we can construct parsers that take $O(n)$ time:
 - Top-down LL parsers for LL grammars (LL = Left-to-right scanning of input, Left-most derivation)
 - Bottom-up LR parsers for LR grammars (LR = Left-to-right scanning of input, Right-most derivation)

Top-Down Parsers and LL Grammars

- *Top-down parser* is a parser for LL class of grammars
 - Also called *predictive parser*
 - LL class is a strict subset of the larger LR class of grammars
 - LL grammars cannot contain *left-recursive productions* (but LR can), for example:
 $\langle X \rangle ::= \langle X \rangle \langle Y \rangle \dots$
and
 $\langle X \rangle ::= \langle Y \rangle \langle Z \rangle \dots$
 $\langle Y \rangle ::= \langle X \rangle \dots$
 - LL(k) where k is lookahead depth, if $k=1$ cannot handle alternatives in productions with common prefixes
 $\langle X \rangle ::= a b \dots \mid a c \dots$
- A top-down parser constructs a parse tree from the root down
- Not too difficult to implement a predictive parser for an unambiguous LL(1) grammar in BNF by hand using *recursive descent*

Top-Down Parser in Action

```

<id_list> ::= id <id_list_tail>
<id_list_tail> ::= , id <id_list_tail>
                | ;
    
```

A, B, C;

<id list>

A, B, C;

```

<id list>
 /  \
A    <id list tail>
    
```

A, B, C;

```

<id list>
 /  \
A    <id list tail>
     /  \
    ,    <id list tail>
     B
    
```

A, B, C;

```

<id list>
 /  \
A    <id list tail>
     /  \
    ,    <id list tail>
     B    /  \
         ,    <id list tail>
         C    ;
    
```

Top-Down Predictive Parsing

- Top-down parsing is called predictive parsing because parser “predicts” what it is going to see:
 1. As root, the start symbol of the grammar $\langle id_list \rangle$ is predicted
 2. After reading **A** the parser predicts that $\langle id_list_tail \rangle$ must follow
 3. After reading **, and B** the parser predicts that $\langle id_list_tail \rangle$ must follow
 4. After reading **, and C** the parser predicts that $\langle id_list_tail \rangle$ must follow
 5. After reading **;** the parser stops

An Ambiguous Non-LL Grammar for Language E

- Consider a language E of simple expressions composed of $+$, $-$, $*$, $/$, $()$, id , and num

$$\begin{aligned} \langle expr \rangle & ::= \langle expr \rangle + \langle expr \rangle \\ & | \langle expr \rangle - \langle expr \rangle \\ & | \langle expr \rangle * \langle expr \rangle \\ & | \langle expr \rangle / \langle expr \rangle \\ & | (\langle expr \rangle) \\ & | \langle id \rangle \\ & | \langle num \rangle \end{aligned}$$

- Need operator precedence rules

An Unambiguous Non-LL Grammar for Language *E*

$$\begin{aligned} \langle expr \rangle & ::= \langle expr \rangle + \langle term \rangle \\ & \quad | \langle expr \rangle - \langle term \rangle \\ & \quad | \langle term \rangle \\ \langle term \rangle & ::= \langle term \rangle * \langle factor \rangle \\ & \quad | \langle term \rangle / \langle factor \rangle \\ & \quad | \langle factor \rangle \\ \langle factor \rangle & ::= (\langle expr \rangle) \\ & \quad | \langle id \rangle \\ & \quad | \langle num \rangle \end{aligned}$$

An Unambiguous LL(1) Grammar for Language *E*

$\langle expr \rangle$	$::= \langle term \rangle \langle term_tail \rangle$
$\langle term \rangle$	$::= \langle factor \rangle \langle factor_tail \rangle$
$\langle term_tail \rangle$	$::= \langle add_op \rangle \langle term \rangle \langle term_tail \rangle$ ϵ
$\langle factor \rangle$	$::= (\langle expr \rangle)$ $\langle id \rangle$ $\langle num \rangle$
$\langle factor_tail \rangle$	$::= \langle mult_op \rangle \langle factor \rangle \langle factor_tail \rangle$ ϵ
$\langle add_op \rangle$	$::= + \mid -$
$\langle mult_op \rangle$	$::= * \mid /$

Constructing Recursive Descent Parsers for LL(1)

- Each nonterminal has a function that implements the production(s) for that nonterminal
- The function parses only the part of the input described by the nonterminal

$\langle expr \rangle ::= \langle term \rangle \langle term_tail \rangle$

```
procedure expr()  
  term(); term_tail();
```

- When more than one alternative production exists for a nonterminal, the lookahead token should help to decide which production to apply

$\langle term_tail \rangle ::= \langle add_op \rangle \langle term \rangle \langle term_tail \rangle$
 | ϵ

```
procedure term_tail()  
  case (input_token())  
  of '+' or '-': add_op(); term(); term_tail();  
  otherwise: /* no op =  $\epsilon$  */
```


Some Rules to Construct a Recursive Descent Parser

- For every nonterminal with more than one production, find all the tokens that each of the right-hand sides can start with:

$\langle X \rangle ::= \mathbf{a}$	starts with a
b a $\langle Z \rangle$	starts with b
$\langle Y \rangle$	starts with c or d
$\langle Z \rangle \mathbf{f}$	starts with e or f
$\langle Y \rangle ::= \mathbf{c} \mid \mathbf{d}$	
$\langle Z \rangle ::= \mathbf{e} \mid \varepsilon$	

- Empty productions are coded as “skip” operations (nops)
- If a nonterminal does not have an empty production, the function should generate an error if no token matches

Example for E

```
procedure expr()  
  term(); term_tail();
```

```
procedure term_tail()  
  case (input_token())  
  of '+' or '-': add_op(); term(); term_tail();  
  otherwise: /* no op =  $\epsilon$  */
```

```
procedure term()  
  factor(); factor_tail();
```

```
procedure factor_tail()  
  case (input_token())  
  of '*' or '/': mult_op(); factor(); factor_tail();  
  otherwise: /* no op =  $\epsilon$  */
```

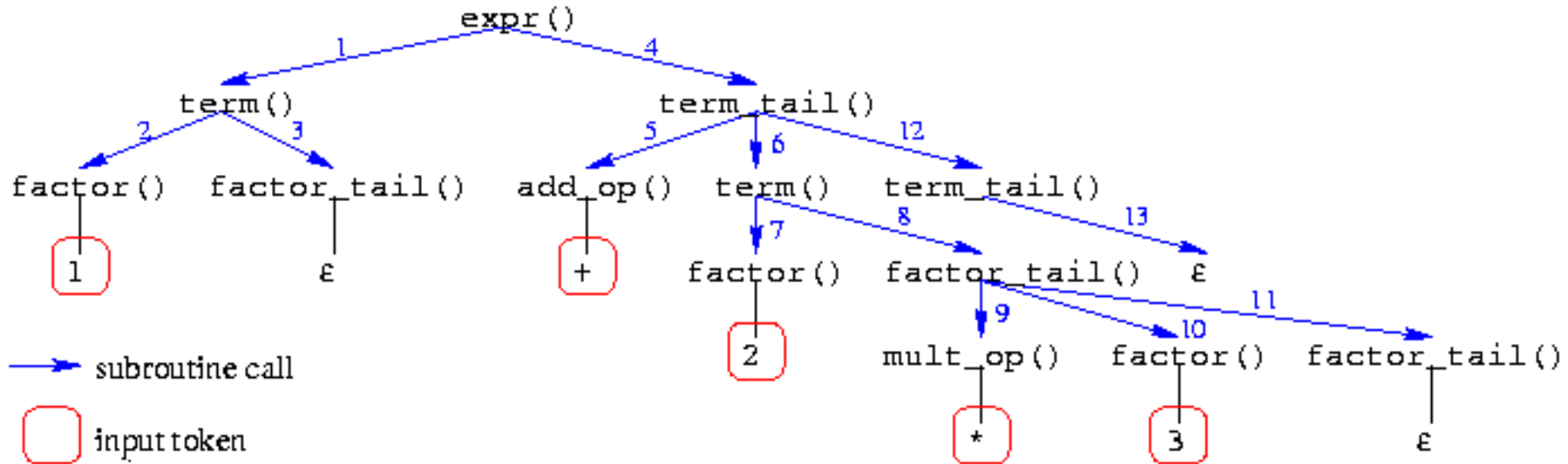
```
procedure factor()  
  case (input_token())  
  of '(': match('('); expr(); match(')');  
  of identifier: match(identifier);  
  of number: match(number);  
  otherwise: error;
```

```
procedure add_op()  
  case (input_token())  
  of '+': match('+');  
  of '-': match('-');  
  otherwise: error;
```

```
procedure mult_op()  
  case (input_token())  
  of '*': match('*');  
  of '/': match('/');  
  otherwise: error;
```

Recursive Descent Parser's Call Graph = Parse Tree

- The *dynamic call graph* of a recursive descent parser corresponds exactly to the parse tree
- Call graph of input string **1+2*3**



Example

<type> ::= *<simple>*
 / ^ **id**
 / **array [*<simple>*] of *<type>***
<simple> ::= **integer**
 / **char**
 / **num dotdot num**

Example (cont'd)

<type> ::= *<simple>*
 / **^ id**
 / **array [*<simple>*] of *<type>***
<simple> ::= **integer**
 / **char**
 / **num dotdot num**

<type> starts with **^** or **array** or anything that *<simple>* starts with
<simple> starts with **integer**, **char**, and **num**

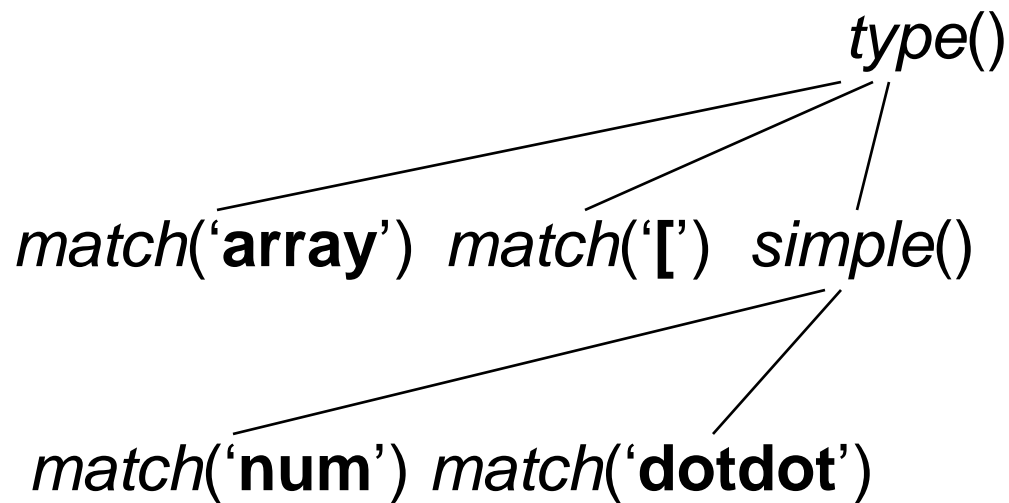
Example (cont'd)

```
procedure match(t : token)  
  if input_token() = t then  
    nexttoken();  
  else error;
```

```
procedure type()  
  case (input_token())  
  of 'integer' or 'char' or 'num':  
    simple();  
  of '^':  
    match('^'); match(id);  
  of 'array':  
    match('array'); match('['); simple();  
    match(']'); match('of'); type();  
  otherwise: error;
```

```
procedure simple()  
  case (input_token())  
  of 'integer':  
    match('integer');  
  of 'char':  
    match('char');  
  of 'num':  
    match('num');  
    match('dotdot');  
    match('num');  
  otherwise: error;
```

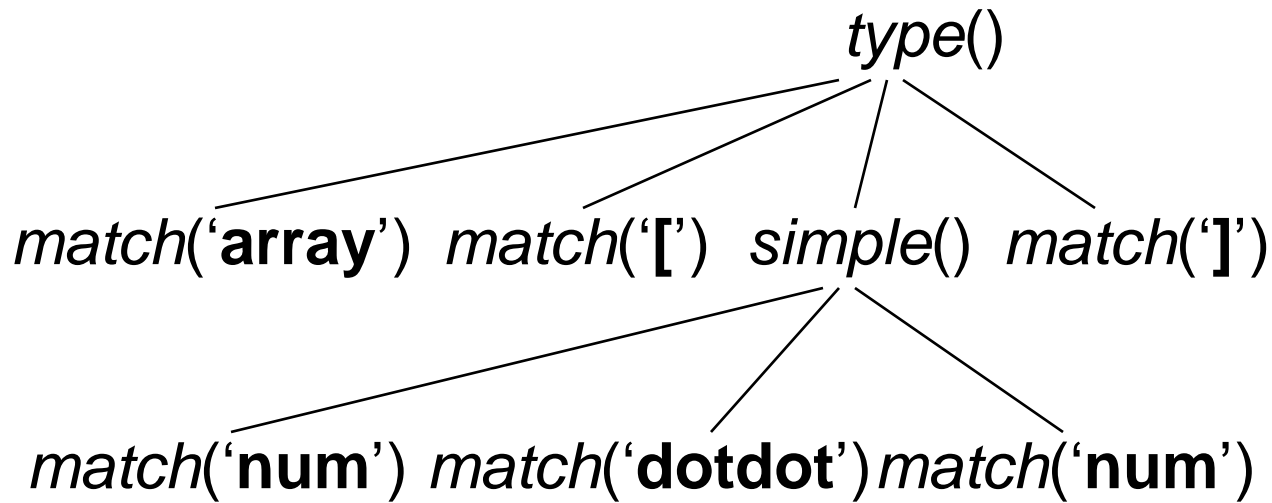

Step 4



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

↑
lookahead

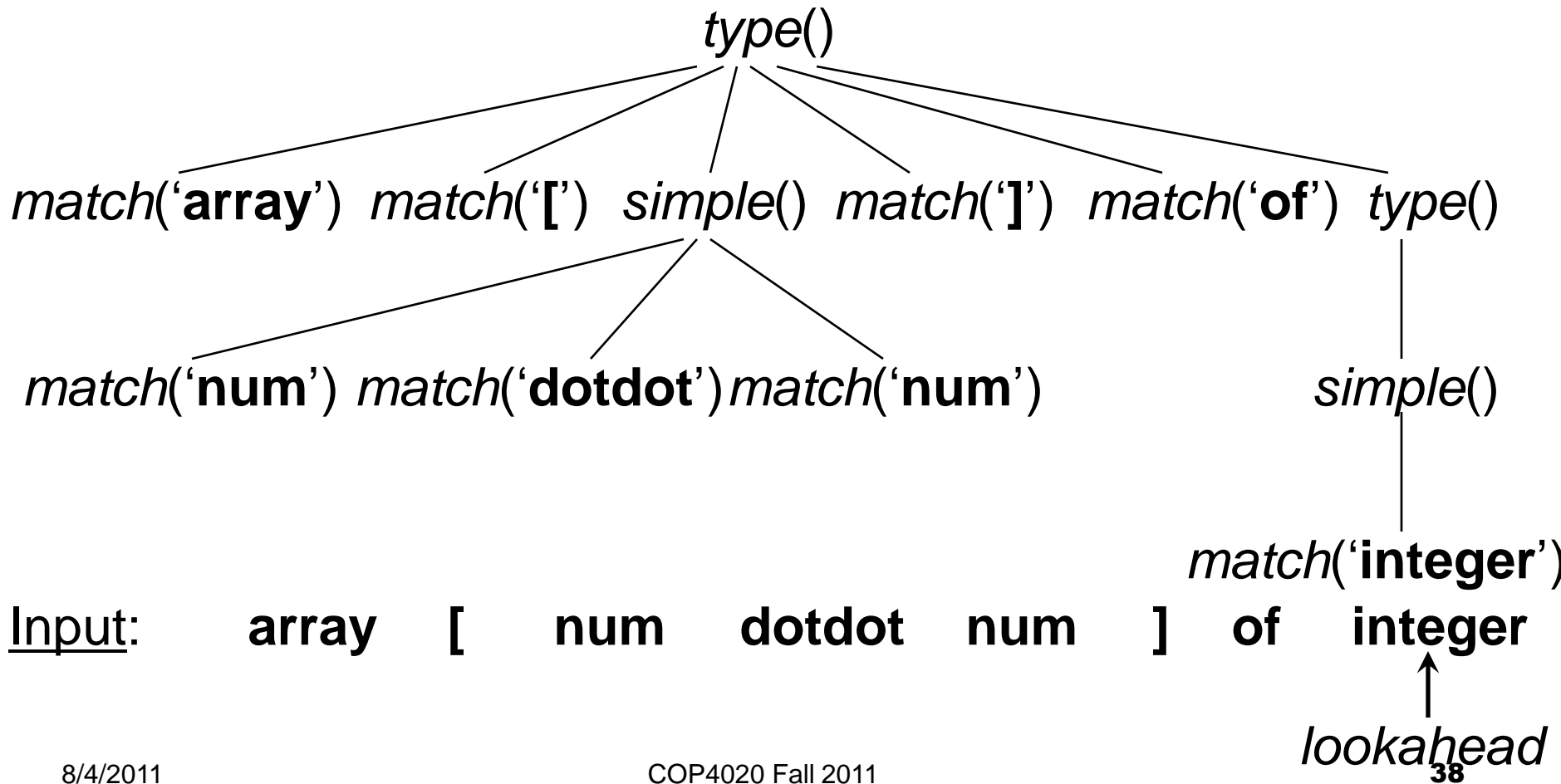
Step 6



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

↑
lookahead

Step 8



Bottom-Up LR Parsing

- Bottom-up parser is a parser for LR class of grammars
- Difficult to implement by hand
- Tools (e.g. Yacc/Bison) exist that generate bottom-up parsers for LALR grammars automatically
- LR parsing is based on shifting tokens on a stack until the parser recognizes a right-hand side of a production which it then reduces to a left-hand side (nonterminal) to form a partial parse tree

Bottom-Up Parser in Action

$\langle id_list \rangle ::= id \langle id_list_tail \rangle$ $\langle id_list_tail \rangle ::= , id \langle id_list_tail \rangle$ $\quad \quad \quad \quad ;$

input	stack	parse tree
A, B, C;	A	
A, B, C;	A,	
A, B, C;	A, B	
A, B, C;	A, B,	
A, B, C;	A, B, C	
A, B, C;	A, B, C;	
A, B, C;	A, B, C	$\langle id_list_tail \rangle$ $\quad \quad \quad $ $\quad \quad \quad ;$

A, B, C;	A, B, C	<pre> <id_list_tail> ; </pre>
A, B, C;	A, B	<pre> <id_list_tail> / \ , <id_list_tail> \ ; ; </pre>
A, B, C;	A	<pre> <id_list_tail> / \ , <id_list_tail> \ / \ ; , <id_list_tail> ; ; </pre>
A, B, C;		<pre> <id_list> / \ A <id_list_tail> \ / \ ; , <id_list_tail> ; ; </pre>