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

    $r_1\, r_2\, r_3 \ldots r_n$
  - *Alternation*: regular expressions separated by a bar

    $r_1 \mid r_2$
  - *Repetition*: a regular expression followed by a star (Kleene star)

    r^*

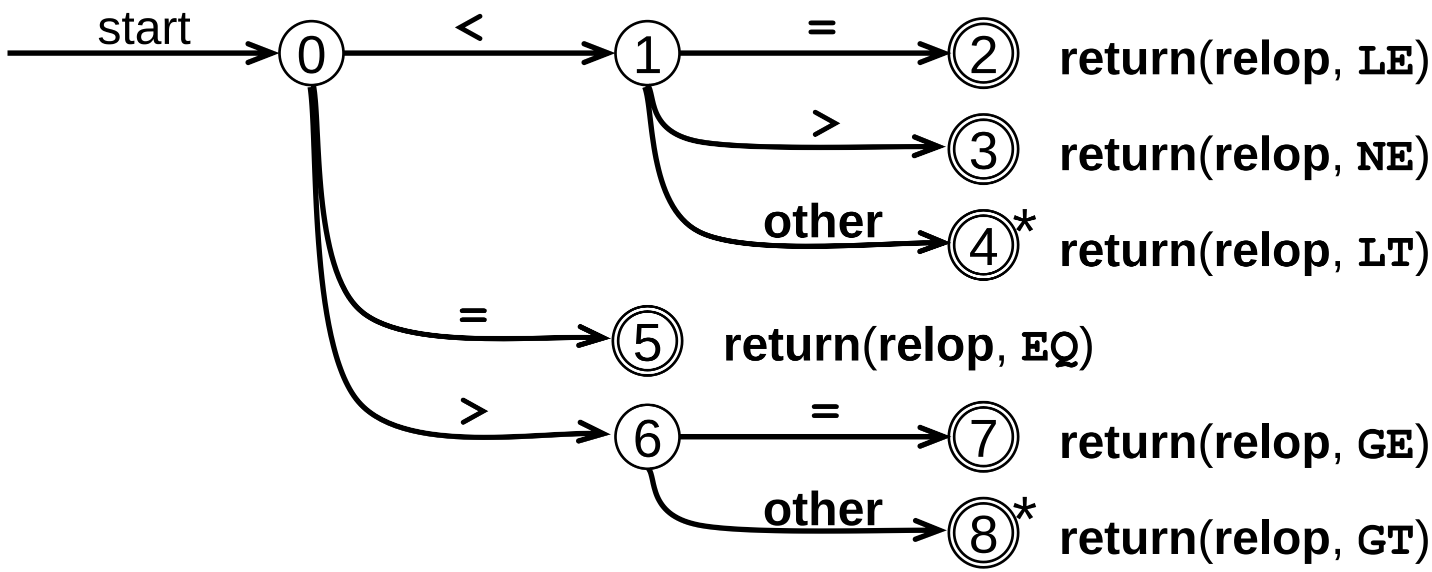
Example Regular Definitions for Tokens

- *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- *unsigned_integer* → *digit digit**
- *signed_integer* → (+ | - | ε) *unsigned_integer*
- *letter* → a | b | … | z | A | B | … Z
- *identifier* → *letter* (*letter* | *digit*)*
- Cannot use recursive definitions! This is illegal:

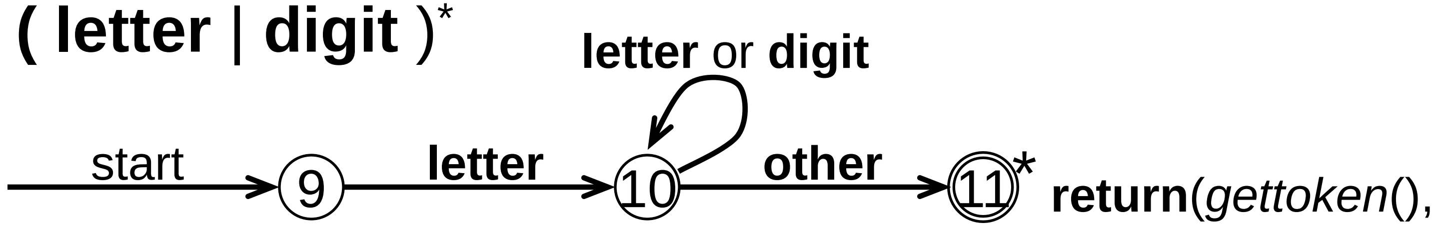
  digits → *digit digits* | *digit*

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

  <*nonterminal*> ::= *sequence of (non)terminals*

  where
  - A *terminal* of the grammar is a *token*
  - A <*nonterminal*> 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

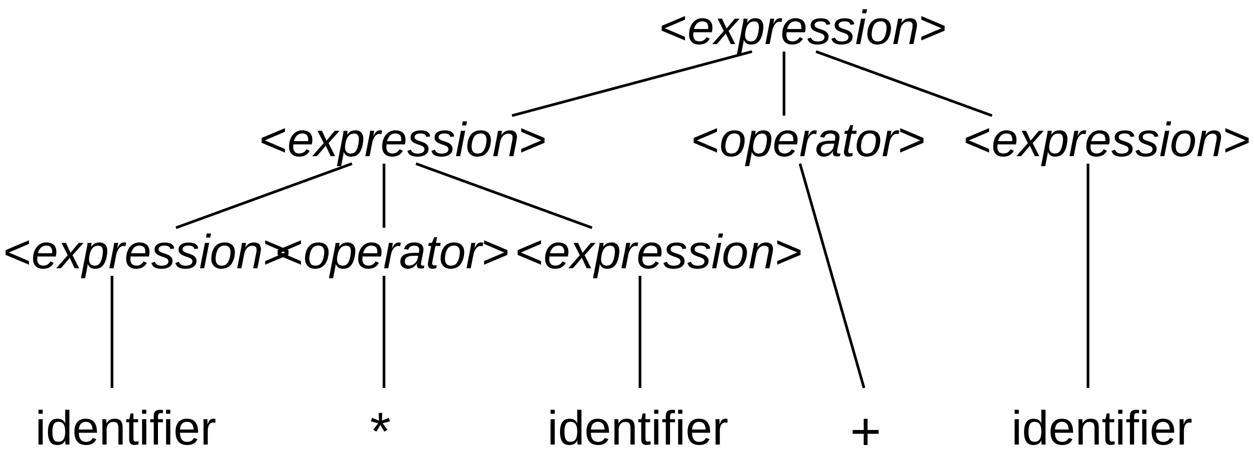
Example

<*expression*> ::= identifier
| unsigned_integer
| **-** <*expression*>
| **(** <*expression*> **)**
| <*expression*> <*operator*> <*expression*>
<*operator*> ::= **+** | **-** | ***** | **/**

<*expression*>
⇒ <*expression*> <*operator*> <*expression*>
⇒ <*expression*> <*operator*> identifier
⇒ <*expression*> + identifier
⇒ <*expression*> <*operator*> <*expression*> + identifier
⇒ <*expression*> <*operator*> identifier + identifier
⇒ <*expression*> * identifier + identifier
⇒ identifier * identifier + identifier

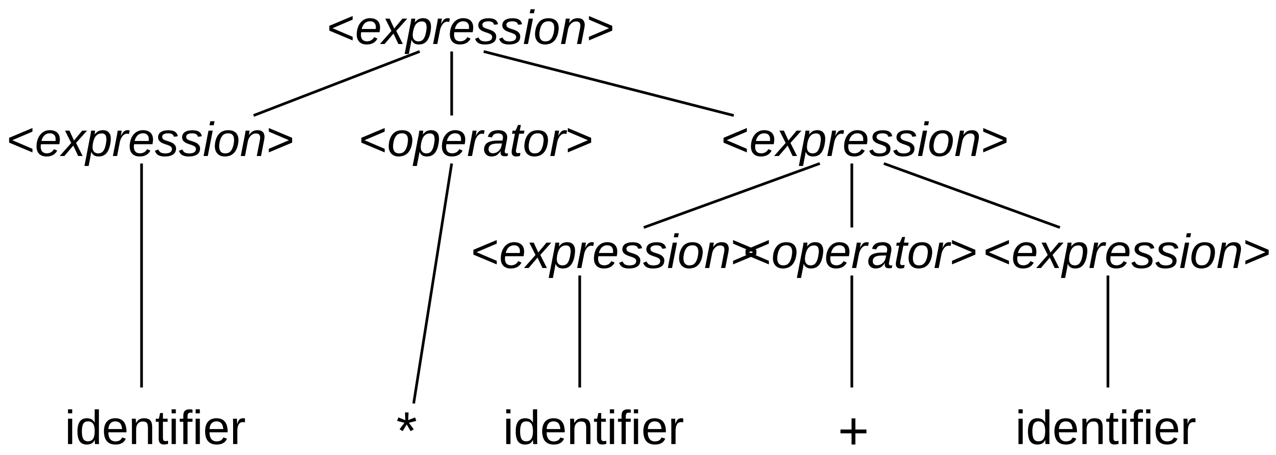
Parse Trees

- 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

<expression>	::= *<term>* | *<expression>* *<add_op>* *<term>*
<term>	::= *<factor>* | *<term>* *<mult_op>* *<factor>*
<factor>	::= identifier | unsigned_integer | **-** *<factor>* | **(** *<expression>* **)**
<add_op>	::= **+** | **-**
<mult_op>	::= ***** | **/**

Ambiguous if-then-else

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

  <*stmt*> ::= **if** <*expr*> **then** <*stmt*>
  | **if** <*expr*> **then** <*stmt*> **else** <*stmt*>

- 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:

  <*stmt*> ::= **if** <*expr*> **then** <*stmt*> **end if**
  | **if** <*expr*> **then** <*stmt*> **else** <*stmt*> **end if**

Linear-Time Top-Down and Bottom-Up Parsing

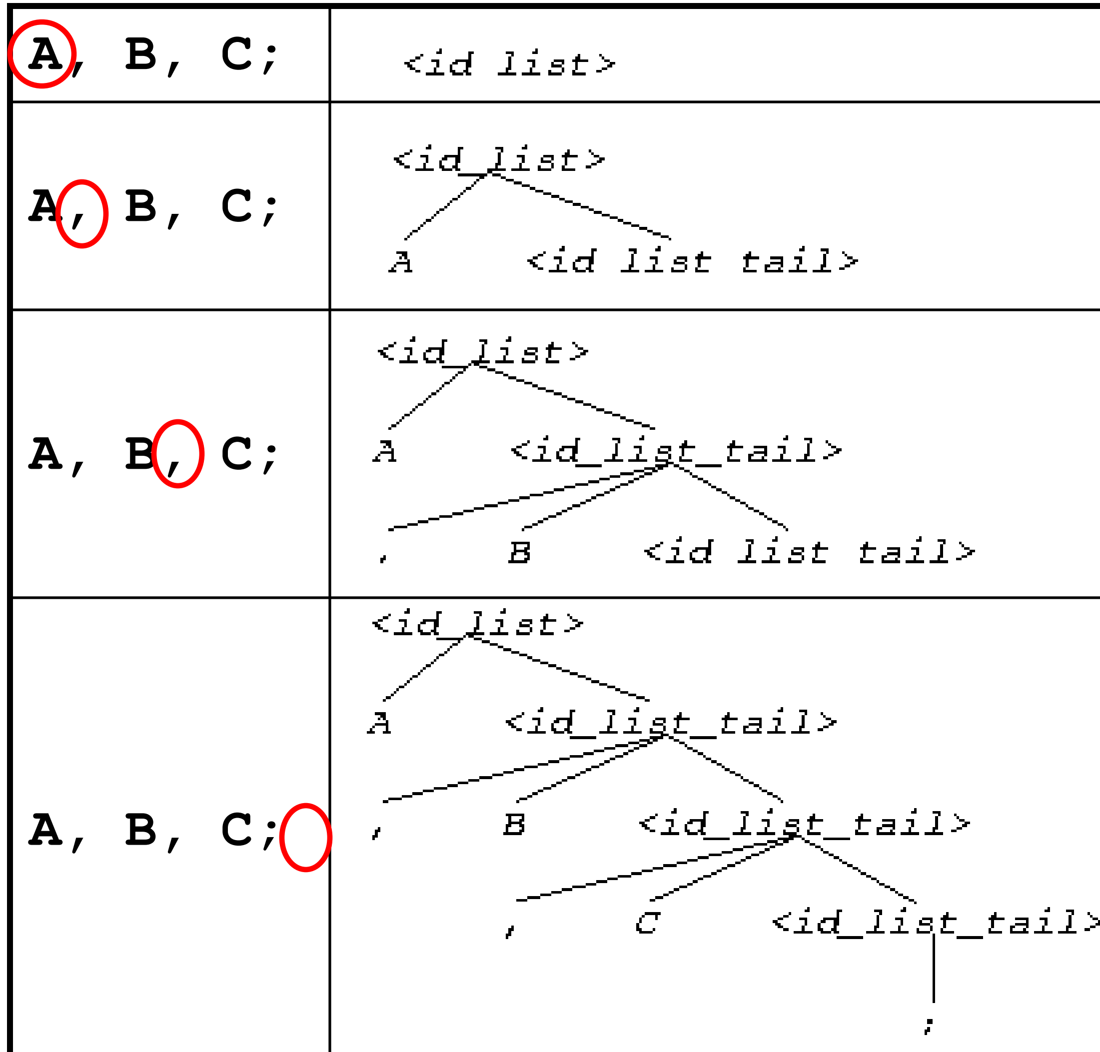
- A *parser* is a recognizer for a context-free language
- A string (token sequence) is accepted by the parser and a parse tree can be constructed if the string is in the language
- For any arbitrary context-free grammar parsing can take as much as $O(n^3)$ time, where n is the size of the input
- There 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:
    <X> ::= <X> <Y> …
    and
    <X> ::= <Y> <Z> …
    <Y> ::= <X> …
  - LL(*k*) where *k* is lookahead depth, if *k*=1 cannot handle alternatives in productions with common prefixes
    <X> ::= a b … | a c …
- 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*>
| **;**



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 <*id_list*> is predicted
  2. After reading **`A`** the parser predicts that <*id_list_tail*> must follow
  3. After reading **`,`** and **`B`** the parser predicts that <*id_list_tail*> must follow
  4. After reading **`,`** and **`C`** the parser predicts that <*id_list_tail*> 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

<expr> ::= *<expr>* + *<expr>*
| *<expr>* - *<expr>*
| *<expr>* * *<expr>*
| *<expr>* / *<expr>*
| (*<expr>*)
| *<id>*
| *<num>*

- Need operator precedence rules

An Unambiguous Non-LL Grammar for Language *E*

<expr> ::= *<expr>* + *<term>*
| *<expr>* - *<term>*
| *<term>*
<term> ::= *<term>* * *<factor>*
| *<term>* / *<factor>*
| *<factor>*
<factor> ::= (*<expr>*)
| *<id>*
| *<num>*

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

<expr>	::= *<term>* *<term_tail>*
<term>	::= *<factor>* *<factor_tail>*
<term_tail>	::= *<add_op>* *<term>* *<term_tail>*
	| ε
<factor>	::= (*<expr>*)
	| *<id>*
	| *<num>*
<factor_tail>	::= *<mult_op>* *<factor>* *<factor_tail>*
	| ε
<add_op>	::= + | -
<mult_op>	::= * | /

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

<expr> ::= *<term>* *<term_tail>*

```
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

<term_tail> ::= *<add_op>* *<term>* *<term_tail>*
| ε

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

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:

<*X*> ::= **a**	starts with **a**
| **b a** <*Z*>	starts with **b**
| <*Y*>	starts with **c** or **d**
| <*Z*> **f**	starts with **e** or **f**

<*Y*> ::= **c** | **d**

<*Z*> ::= **e** | ε

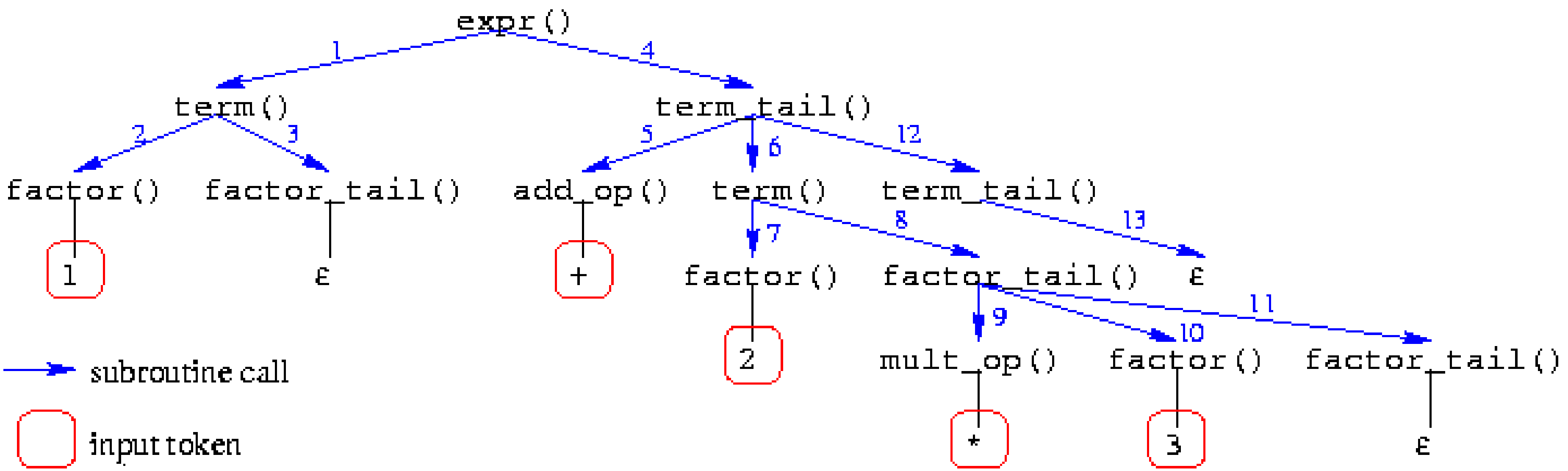
- Empty productions are coded as “skip” operations (nops)
- If a nonterminal does not have an empty production, the function should generate an error if no token matches

Example for *E*

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

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

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

procedure factor_tail()
  case (input_token())
  of '*' or '/': mult_op(); factor(); factor_tail();
  otherwise: /* no op = ε */

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)

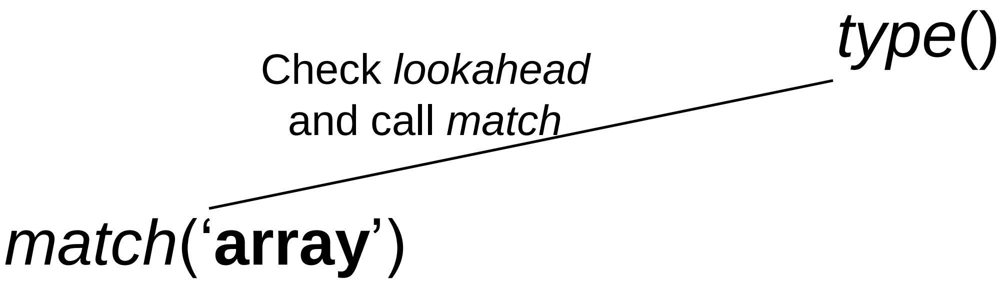
<*type*> ::= <*simple*>
| **^ id**
| **array [** <*simple*> **] of** <*type*>
<*simple*> ::= **integer**
| **char**
| **num dotdot num**

<*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 1



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

↑

lookahead

Step 2

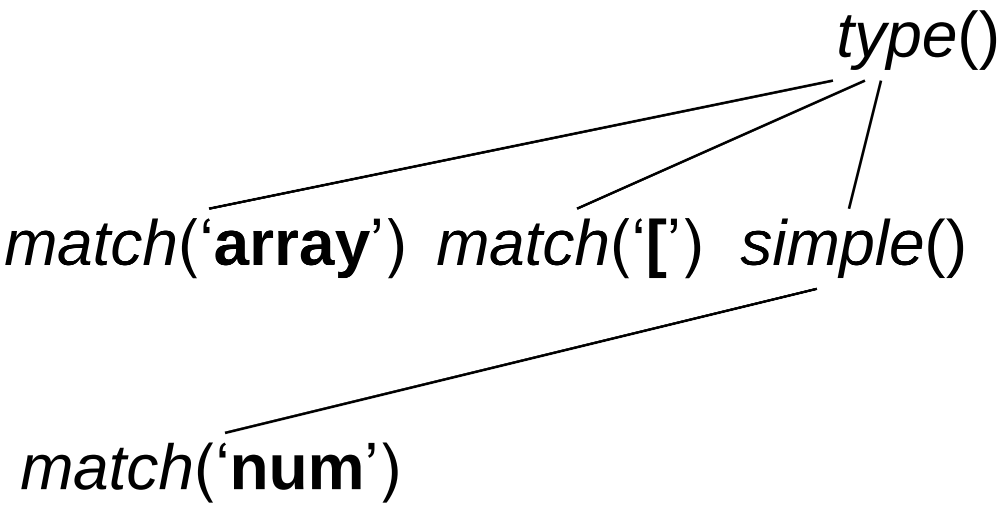
type()

match(‘**array**’) *match*(‘**[**’)

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

lookahead

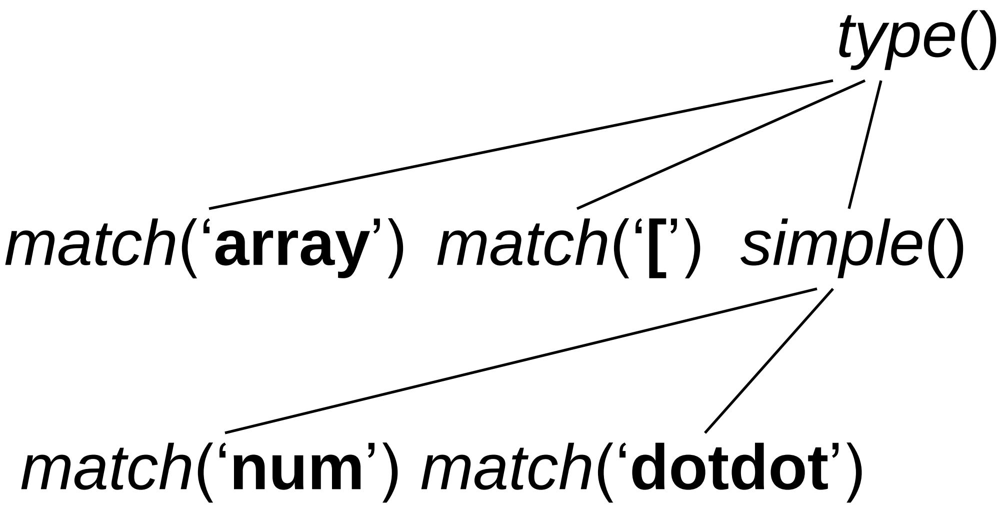
Step 3



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

↑ *lookahead*

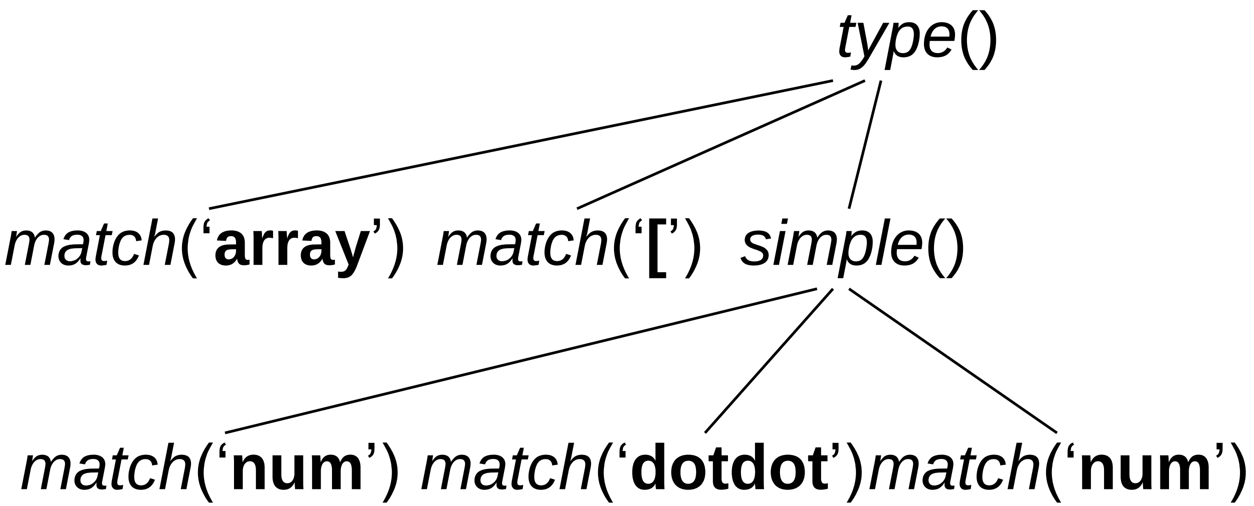
Step 4



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

↑ *lookahead*

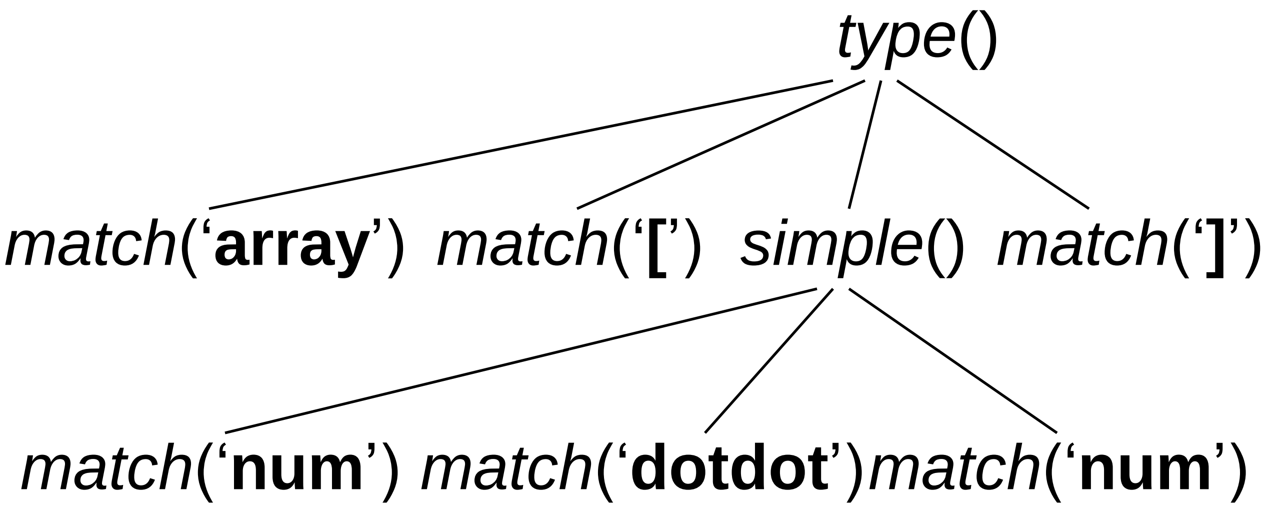
Step 5



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

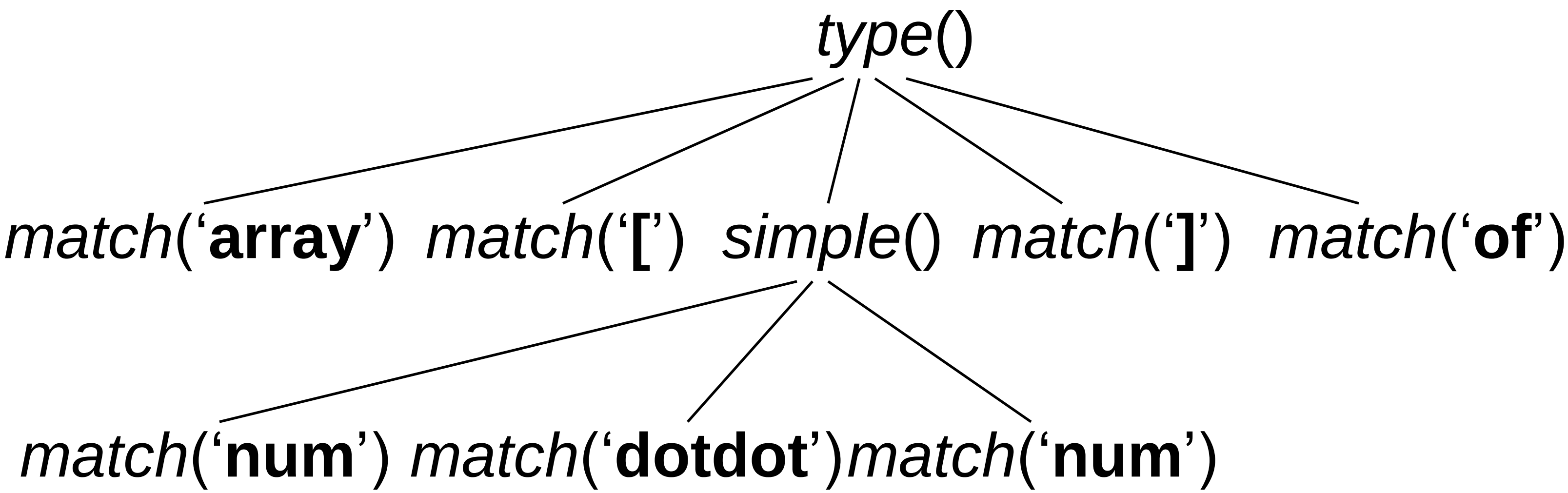
↑ *lookahead*

Step 6



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

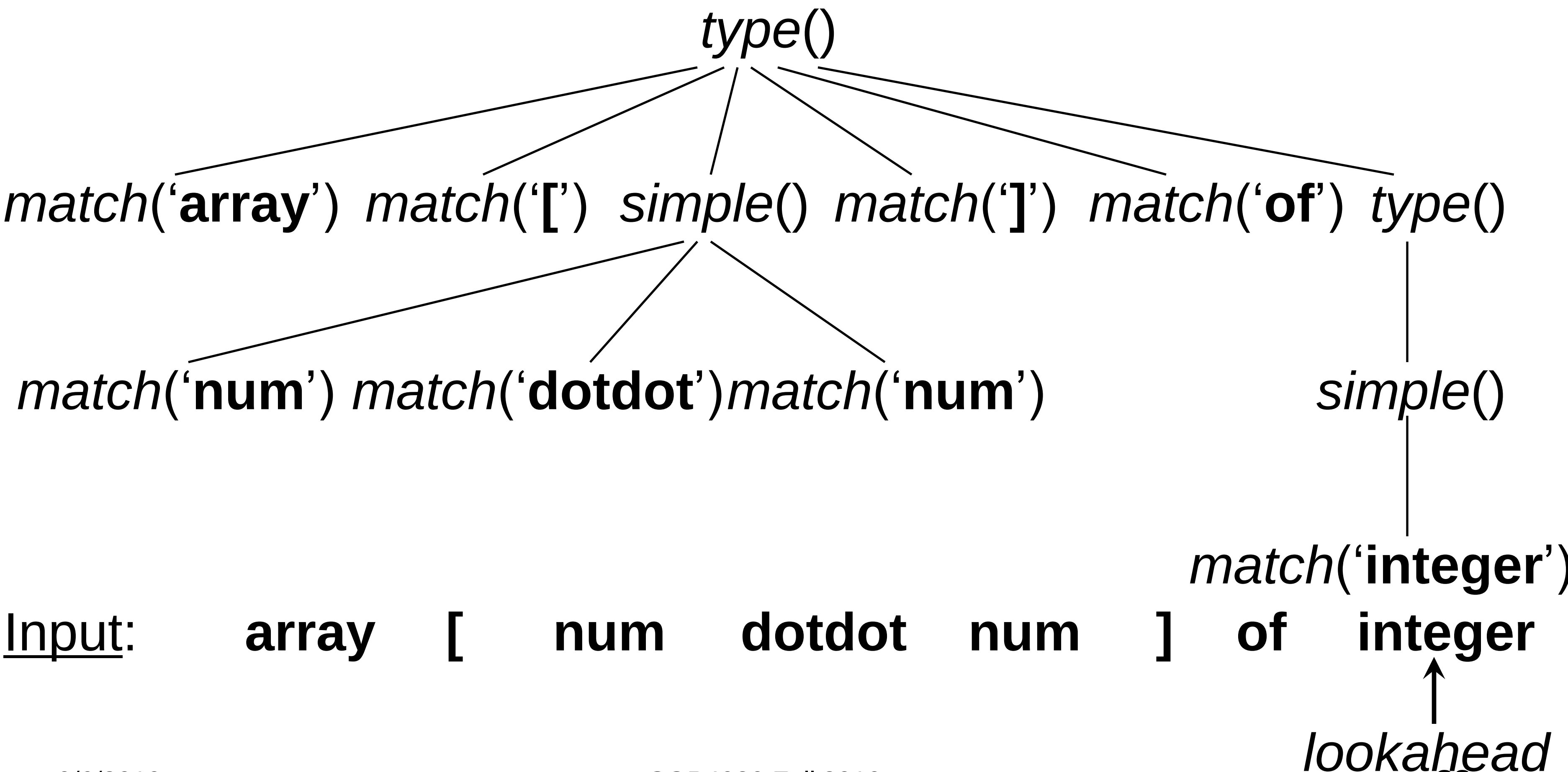
Step 7



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

↑ *lookahead*

Step 8



type()
match('array') match('[') simple() match(']') match('of') type()
match('num') match('dotdot') match('num') simple()
match('integer')
Input: array [num dotdot num] of integer
lookahead

Bottom-Up LR Parsing

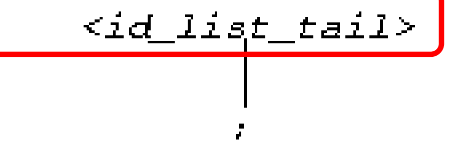
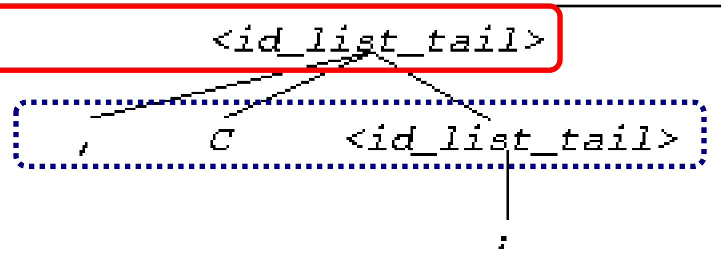
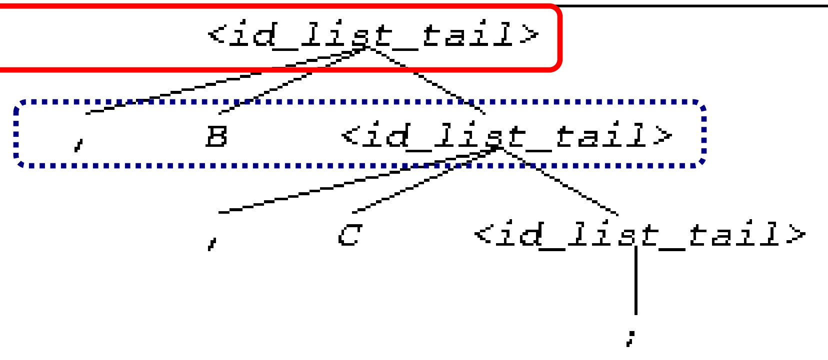
- Bottom-up parser is a parser for LR class of grammars
- Difficult to implement by hand
- Tools (e.g. Yacc/Bison) exist that generate bottom-up parsers for LALR grammars automatically
- LR parsing is based on shifting tokens on a stack until the parser recognizes a right-hand side of a production which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

Bottom-Up Parser in Action

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

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**	

Cont'd …

A, B, C;	**A,B,C**	
A, B, C;	**A,B**	
A, B, C;	**A**	
A, B, C;