CSE 131A
Lecture 19
Top-Down Parsing

Scott B. Baden
Department of Computer Science & Engineering
University of California, San Diego
Winter 2007

Announcements

- Wednesday's office hours are cancelled

Today's Lecture

- Time costs of different types of parsers
- Top-down parsing

Different types of parsers

- Bottom-up parsers: build a parse tree starting from leaves and working up to the root
  - Shift-reduce parsing, LR parsing
  - Often used in production compilers
  - Built with a parser generator
- Top-down parsers: build parse tree starting from root, and working down to leaves
  - Predictive recursive descent parsing, LL parsing
- Universal parsers: can parse any CFL
  - Cocke-Younger-Kasami, Earley algorithms

Parser time costs

- A parser can be constructed for any context-free grammar, since membership in a CFL is decidable
- CFL membership is in the complexity class **P**
  - For any context free grammar, there is a parser that runs in $O(n^3)$ time in the worst case, where n is the length of the token sequence given as input
- Universal context-free parsers such as Earley, Cocke-Younger-Kasimi are $O(n^3)$ parsers
- But $O(n^3)$-time parsers are too slow for use in practical programming language compilers…

Parser time costs

- Fortunately, it is not too difficult to design programming languages with syntax that can be parsed more efficiently
- LR(1) parsers
  - Running time $O(n)$ in the worst case
  - single pass through input, only 1-token lookahead
- LL(1) top down recursive-descent parsers
  - also run in $O(n)$-time
  - today, we'll discuss this type of parser

LL Parsers

- LL parsers
  - Perform a Left-to-right scan of the input
  - Trace a Leftmost derivation (NOT in reverse)

- We'll look at the design of a procedural, top-down LL predictive parser
  - More easily implemented "by hand" than an LR parser
  - Less powerful than LR for the same amount of lookahead

A example grammar (on board)

- Consider the following grammar for a subset of Pascal type declarations:

type ::= *simple*
  | ↑ **id**
  | **array** [*simple*] **of** *type*

simple **::=** **integer**
  | **char**
  | **num** “..” **num**

Example: **array [num .. num] of integer**
where **1** and **10** are **num** tokens

Basic Top-down Parsing "Algorithm"

1. Construct the root of the parse tree, labeled with the start symbol. Initialize the *current node* to be the root, and initialize *current token* to be the first token in the input.
2. If the current node is labeled with a nonterminal, select a production consistent with the current token, and construct child nodes of the current node for the symbols on the RHS of the production.
3. Else if the current node is labeled with a terminal equal to the current token, advance the input, and set the current token to be the next token in the input.
4. If all input has been consumed and all leaves of the parse tree are matched terminals, done! Else select leftmost untouched leaf node of the parse tree as the current node and go to 2.

(NOTE: In general, the choice of rule in 2 might not "work," and *backtracking* may be necessary.)

Building a parse tree top-down

array [num .. num] of integer

type

Root, labeled with start symbol, is current node;
input ready at first token

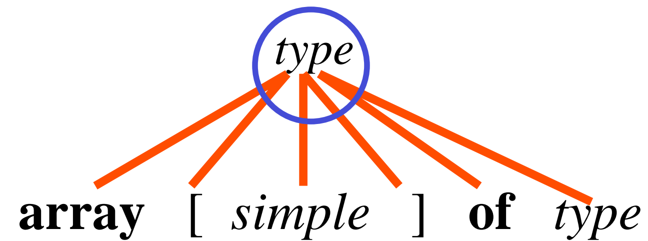
Basic Top-down Parsing "Algorithm"

1. Construct the root of the parse tree, labeled with the start symbol. Initialize the *current node* to be the root, and initialize *current token* to be the first token in the input.
2. If the current node is labeled with a nonterminal, select a production consistent with the current token, and construct child nodes of the current node for the symbols on the RHS of the production.
3. Else if the current node is labeled with a terminal equal to the current token, advance the input, and set the current token to be the next token in the input.
4. If all input has been consumed and all leaves of the parse tree are matched terminals, done! Else select leftmost untouched leaf node of the parse tree as the current node and go to 2.

(NOTE: In general, the choice of rule in 2 might not "work," and *backtracking* may be necessary.)

Building a parse tree top-down

array [num .. num] of integer



Pick a grammar production (rule) for current node that is consistent with lookahead and expand

type → **array** [*simple*] **of** *type*

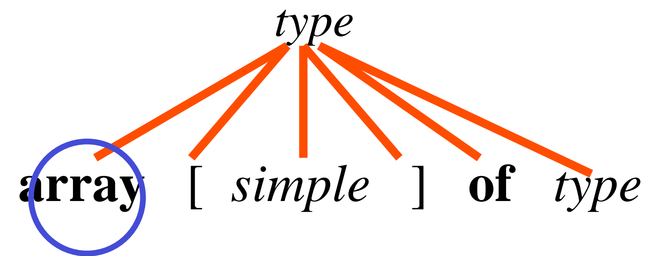
Basic Top-down Parsing "Algorithm"

1. Construct the root of the parse tree, labeled with the start symbol. Initialize the *current node* to be the root, and initialize *current token* to be the first token in the input.
2. If the current node is labeled with a nonterminal, select a production consistent with the current token, and construct child nodes of the current node for the symbols on the RHS of the production.
3. Else if the current node is labeled with a terminal equal to the current token, advance the input, and set the current token to be the next token in the input.
4. If all input has been consumed and all leaves of the parse tree are matched terminals, done! Else select leftmost untouched leaf node of the parse tree as the current node and go to 2.

(NOTE: In general, the choice of rule in 2 might not "work," and *backtracking* may be necessary.)

Building a parse tree top-down

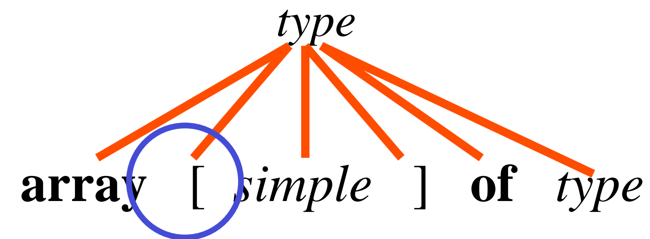
array [num .. num] of integer



Select leftmost untouched leaf node...
array is terminal, so advance input

Building a parse tree top-down

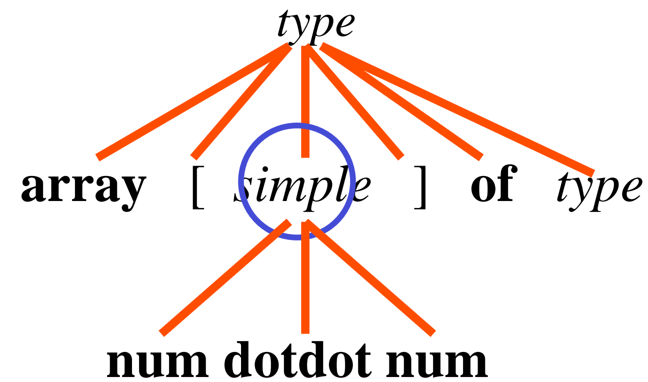
array [num .. num] of integer



Select leftmost untouched leaf node...
[is terminal, so advance input

Building a parse tree top-down

array [num .. num] of integer

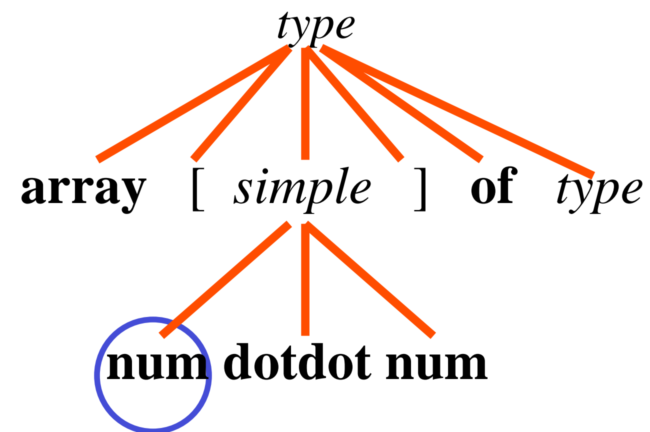


Select leftmost untouched leaf node...
simple is nonterminal, so pick a rule that is consistent with lookahead and expand

simple → **num dotdot num**

Building a parse tree top-down

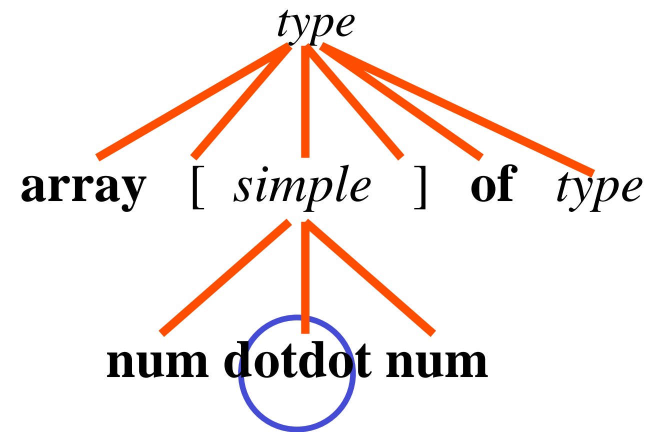
array [num .. num] of integer



Select leftmost untouched leaf node...
num is terminal, so advance input

Building a parse tree top-down

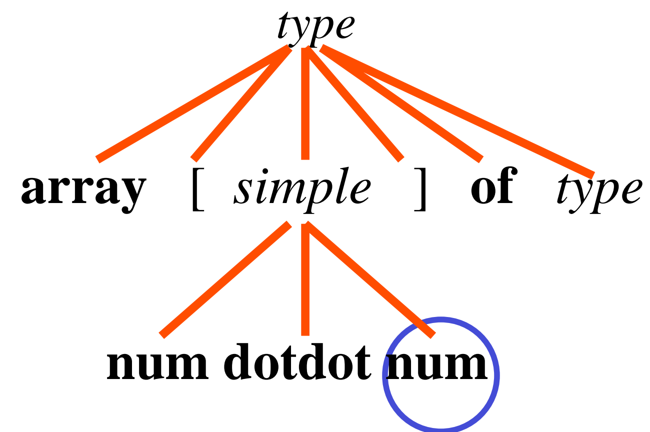
array [num .. num] of integer



Select leftmost untouched leaf node...
dotdot is terminal, so advance input

Building a parse tree top-down

array [num .. num] of integer

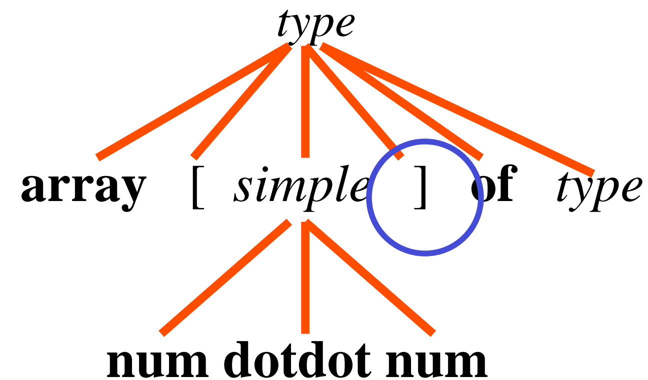


Select leftmost untouched leaf node...

num is terminal, so advance input

Building a parse tree top-down

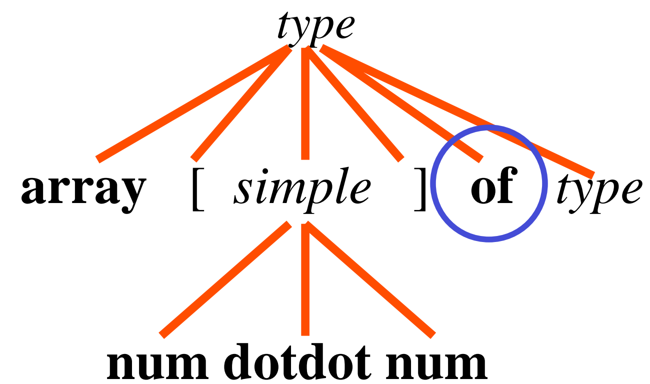
array [num .. num] of integer



Select leftmost untouched leaf node...
] is terminal, so advance input

Building a parse tree top-down

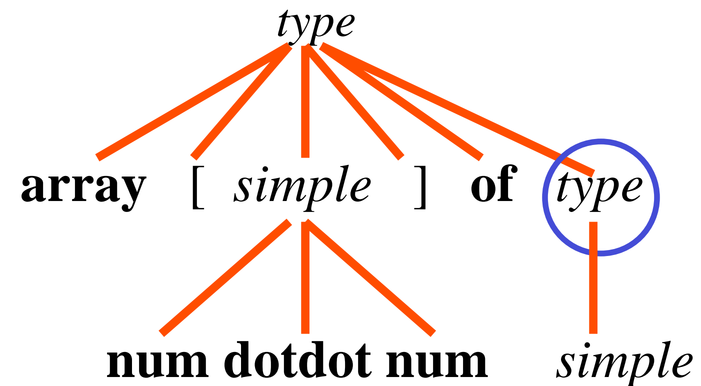
array [num .. num] of integer



Select leftmost untouched leaf node...
of is terminal, so advance input

Building a parse tree top-down

array [num .. num] of integer



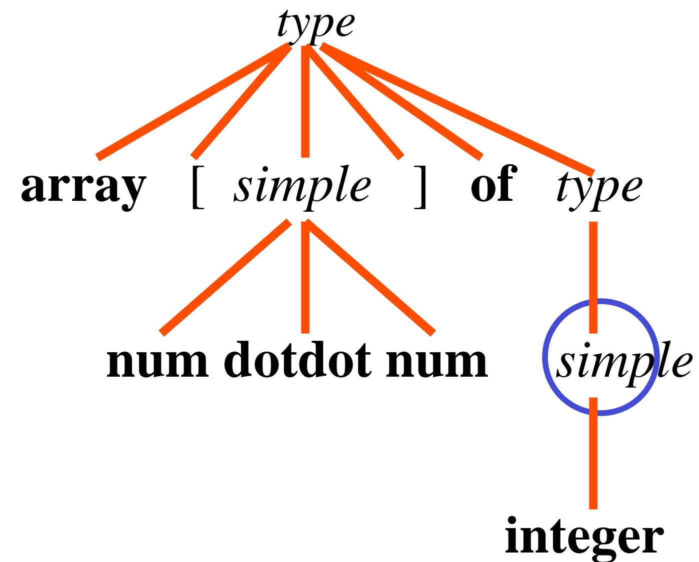
Select leftmost untouched leaf node...
type is nonterminal, so pick a rule that is consistent with lookahead and expand

type → ***simple***

Building a parse tree top-down

array [num .. num] of integer

Select leftmost untouched leaf node...
simple is nonterminal, so pick a rule
that is consistent with lookahead
and expand
simple → **integer**



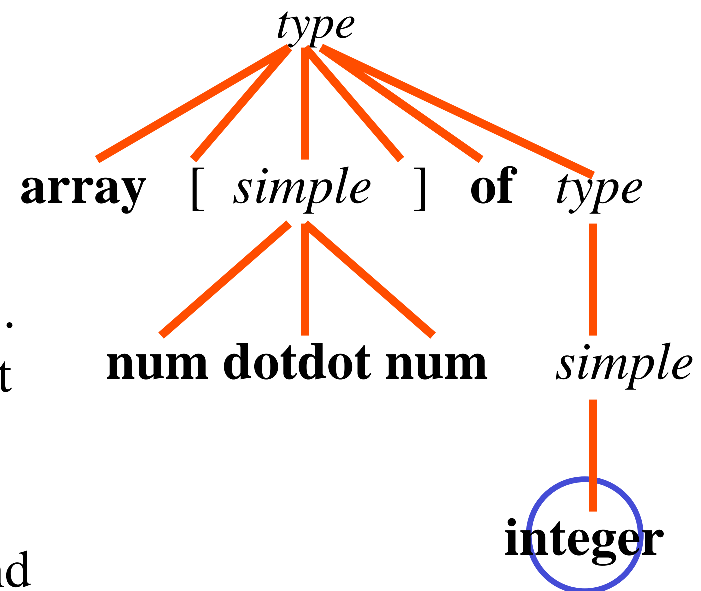
Basic Top-down Parsing "Algorithm"

1. Construct the root of the parse tree, labeled with the start symbol. Initialize the *current node* to be the root, and initialize *current token* to be the first token in the input.
2. If the current node is labeled with a nonterminal, select a production consistent with the current token, and construct child nodes of the current node for the symbols on the RHS of the production.
3. Else if the current node is labeled with a terminal equal to the current token, advance the input, and set the current token to be the next token in the input.
4. If all input has been consumed and all leaves of the parse tree are matched terminals, done! Else select leftmost untouched leaf node of the parse tree as the current node and go to 2.

(NOTE: In general, the choice of rule in 2 might not "work," and *backtracking* may be necessary.)

Building a parse tree top-down

array [num .. num] of integer



Select leftmost untouched leaf node...
integer is terminal, so advance input

Now all input has been consumed and all parse tree leaves are matched terminals... so parse is successful

Leftmost Derivation

- Recall our grammar:

type	::=	*simple* | ↑ **id** | **array** [*simple*] **of** *type*
simple	::=	**integer** | **char** | **num** “..” **num**

- The top-down parse corresponds to a leftmost derivation

type	=>	**array [** *simple* **] of** *type*
	=>	**array [num .. num] of** ***type***
	=>	**array [num .. num] of** ***simple***
	=>	**array [num .. num] of integer**

About that algorithm

- We were lucky; with the particular combination of grammar and input we could easily find a production that permitted us continue on to a successful parse of the input without any mis-steps
- With other CFGs or inputs we might have to "backtrack" around failed a failed parse, considering other productions that might work
- Then we would need to keep track of productions we have tried so we can tell when we have tried them all (if none of them work, input contains a syntax error)
- Better: be sure we always *know* the right production to apply, just looking ahead at the next token
- Only possible with some kinds of context-free grammars

Lookahead grammars

- Suppose a grammar has a production $A \rightarrow \alpha$
- Let the "**lookahead set**" for α, called **First(α)**, be the set of all tokens t such that t appears as the first symbol in some derivation from α
- Of course, some nonterminals may have multiple right-hand-sides:

$$A \rightarrow \alpha \mid \beta \mid \gamma$$

Lookahead grammars

- If for every nonterminal in the grammar, the lookahead sets for its right-hand-sides are disjoint, then it is a 1-symbol lookahead grammar
- Thus, for each non-terminal…
  - We look at all the right hand sides $\alpha_1, \alpha_2, \alpha_n$
  - Compute $\text{First}(\alpha_i) \ \forall \ i$
  - These sets should share nothing in common
- With a 1-symbol lookahead, we will always know what production to apply, given the current input token. No backtracking is needed!

Finding lookahead sets

- Consider the type declaration grammar:

type → *simple*
  | ↑ **id**
  | **array** [*simple*] **of** *type*
simple → **integer**
  | **char**
  | **num** ".." **num**

- Compute these. Is the grammar 1-sym lookahead?

First(*simple*) =
First(↑ **id**) =
First(**array** [*simple*] **of** *type*) =
First(**integer**) =
First(**char**) =
First(**num ".." num**) =

Predictive Parsing

- We'd like to avoid costly backtracking
- With a lookahead grammar, the lookahead sets for a nonterminal let us avoid the guesswork
- This permits *predictive parsing*
- Let's build a simple predictive parser to handle the example Pascal type declaration language
- We will take a procedural approach: we will define a *procedure* for each nonterminal
  - Contrast with table-driven approaches

Design of the predictive parser

- Write a procedure for each nonterminal in the grammar. For our example, they are:
  - *type( )*
  - *simple( )*
- Writing these requires knowing the nonterminals' lookahead sets
- Keep a global variable lookahead indexing the current input token
- Basic idea: if the current lookahead is in the lookahead set of one of this nonterminal's right-hand-sides, then call the procedures for symbols on that RHS in order

Design of the predictive parser

- The parser entry point is the start symbol procedure *type( )*
- A procedure for matching terminals to input tokens: *match(token : t)*
- If the argument token matches the next input token, then the token will be consumed from the input, else an error
- The variable lookahead initially indexes the first token in the input

Predictive parser procedure type()

procedure type
  if lookahead ∈ { **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( )*;
  end if
  else *error()*
end *type*

type → *simple*
  | **↑id**
  | **array** [*simple*] **of** *type*

array [num .. num] **of integer**

Predictive parser procedure match()

- Advances lookahead to the next input token if the argument matches the current lookahead token

procedure match(t:token)
  if (lookahead == t)   **then**
    lookahead := *next_token*( )
  else *error( )*
end match

Predictive parser procedure simple()

simple → **integer**
| **char**
| **num** ".." **num**

procedure simple
if lookahead = **integer then**
match(**integer**)
else if lookahead = **char then**
match(**char**);
else if lookahead = **num then**
***match*(num); *match*(ddot); *match*(num);**
end if
else *error()*
end *type*

array [num .. num] **of integer**

A Top-Down Predictive Parse

- Input tokens **array [num .. num] of integer**
- Lookahead symbol initially points at first token: **array**
- *type( )* checks lookahead, and calls:
  match(array); match([ ); simple( );
  match( ]); match(of); type( );
- This corresponds to the production
  type → **array** [*simple*] **of** *type*

A Top-Down Predictive Parse

- Consume tokens with match(array); match([ );
- Lookahead points to num
- *simple( )* (Afterward, lookahead points to])

  match(num); match(ddot); match(num);
- Consume tokens with match(]); match(of);
- *type( )* is called recursively, calls simple( ); corresponds to the rule type → simple
- *simple( )* returns

  match(integer);
- The call tree exactly corresponds to the parse tree
  - Run time stack is being used as the parser's stack

Top down predictive parsing and left recursion

- As with all recursion, there better be a “base case” that terminates the recursion and permits forward progress
- For our predictive parser, this occurs when the leftmost symbol in the RHS of a production is a terminal that matches the current token
- The token is consumed and we progress toward the end of the input
- But this will not happen if a “left-recursive” production of the form $A \rightarrow A\alpha$ is ever applied

Left recursion and lookahead

- Can a lookahead grammar ever have a left-recursive production?
- Suppose the grammar has a left-recursive production
  $A \rightarrow A\alpha$
- For A to derive sentences at all, the grammar must also have another production on A, say $A \rightarrow \beta$
- But since $A \rightarrow A\alpha \rightarrow \beta\alpha$, the nonterminal A has two productions with overlapping lookahead sets
- So no 1-symbol lookahead grammar will be left-recursive
- But left recursion can be eliminated

Left recursion

- Consider the following production, where the left most symbol on the RHS is the nonterminal being defined
  expr* → *expr* + *term
- If we execute this production, we loop forever, as no input is consumed
- So we can't have left-recursive rules in the grammar for a top-down predictive parser!
- There is always a way to eliminate left recursion without changing the language defined by the grammar, but…
- But we change the syntactic structure

Converting to right recursion

- The left recursive production

$$A \rightarrow A\alpha \mid \beta$$

- Can be rewritten as

$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R \mid \varepsilon$$

- R is a new nonterminal; ε is the empty symbol; and we now have a **right recursive** production $R \rightarrow \alpha R$
- We assume that neither α nor β begins with A, and of course neither α nor ε begin with R
- The new right-recursive grammar can generate exactly the same set of strings as the old one, but note that the parse trees will be different…

Eliminating left recursion

- Applying the notation to the example production:
  $expr \rightarrow expr + term \mid term$
- We set
  $A = expr,\ \alpha = +term,\ \beta = term$
- We introduce a nonterminal R:
  $expr \rightarrow term\ R$
  $R \rightarrow +term\ R \mid \varepsilon$
- A recursive descent parser can handle this grammar
- Different parse tree

$A \rightarrow A\alpha \mid \beta$
$A \rightarrow \beta R$
$R \rightarrow \alpha R \mid \varepsilon$