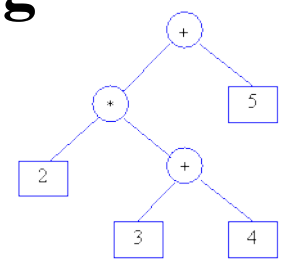
4 (c) parsing

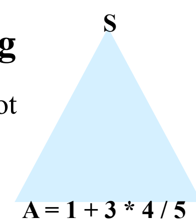


2 * (3 + 4) + 5

Parsing

- A grammar describes syntactically legal strings in a language
- A *recogniser* simply accepts or rejects strings
- A *generator* produces strings
- A *parser* constructs a parse tree for a string
- Two common types of parsers:
  - bottom-up or data driven
  - top-down or hypothesis driven
- A *recursive descent parser* easily implements a top-down parser for simple grammars

Top down vs. bottom up parsing



- The parsing problem is to connect the root node S with the tree leaves, the input
- **Top-down parsers:** starts constructing the parse tree at the top (root) and move down towards the leaves. Easy to implement by hand, but requires restricted grammars. E.g.:
  - Predictive parsers (e.g., LL(k))
- **Bottom-up parsers:** build nodes on the bottom of the parse tree first. Suitable for automatic parser generation, handles larger class of grammars. E.g.:
  - shift-reduce parser (or LR(*k*) parsers)
- Both are general techniques that can be made to work for all languages (but not all grammars!).

Top down vs. bottom up parsing

- Both are general techniques that can be made to work for all languages (but not all grammars!)
- Recall that a given language can be described by several grammars
- Both of these grammars describe the same language

```
E -> E + Num
E -> Num
```

```
E -> Num + E
E -> Num
```

- The first one, with it's left recursion, causes problems for top down parsers
- For a given parsing technique, we may have to transform the grammar to work with it

Parsing complexity

- How hard is the parsing task? How to we measure that?
- Parsing an arbitrary CFG is $O(n^3)$ -- it can take time proportional the cube of the number of input symbols
  - This is bad! (why?)
- If we constrain the grammar somewhat, we can always parse in linear time. This is good! (why?)
- Linear-time parsing
  - LL parsers
    - Recognize LL grammar
    - Use a top-down strategy
  - LR parsers
    - Recognize LR grammar
    - Use a bottom-up strategy

- **LL(n) : Left to right, Leftmost derivation, look ahead at most n symbols.**
- **LR(n) : Left to right, Right derivation, look ahead at most n symbols.**

Top Down Parsing Methods

- Simplest method is a full-backup, *recursive descent* parser
- Often used for parsing simple languages
- Write recursive recognizers (subroutines) for each grammar rule
  - If rules succeeds perform some action (i.e., build a tree node, emit code, etc.)
  - If rule fails, return failure. Caller may try another choice or fail
  - On failure it "backs up"

Top Down Parsing Methods: Problems

- When going forward, the parser consumes tokens from the input, so what happens if we have to back up?
  - suggestions?
- Algorithms that use backup tend to be, in general, inefficient
  - There might be a large number of possibilities to try before finding the right one or giving up
- Grammar rules which are left-recursive lead to non-termination!

Recursive Decent Parsing: Example

For the grammar:

```
<term> -> <factor> {(*|/)<factor>}*
```

We could use the following recursive descent parsing subprogram (this one is written in C)

```
void term() {
  factor();      /* parse first factor*/
  while (next_token == ast_code ||
         next_token == slash_code) {
    lexical();   /* get next token */
    factor();    /* parse next factor */
  }
}
```

Problems

- Some grammars cause problems for top down parsers
- Top down parsers do not work with left-recursive grammars
  - E.g., one with a rule like: E -> E + T
  - We can transform a left-recursive grammar into one which is not
- A top down grammar can limit backtracking if it only has one rule per non-terminal
  - The technique of rule factoring can be used to eliminate multiple rules for a non-terminal

Left-recursive grammars

- A grammar is left recursive if it has rules like

```
X -> X β
```

- Or if it has indirect left recursion, as in

```
X -> A β
A -> X
```

- Q: Why is this a problem?
  - A: it can lead to non-terminating recursion!

Left-recursive grammars

- Consider

```
E -> E + Num
E -> Num
```

- We can manually or automatically rewrite a grammar removing left-recursion, making it ok for a top-down parser.

Elimination of Left Recursion

- Consider left-recursive grammar

```
S → S α
S -> β
```

- S generates strings
  β
  β α
  β α α …
- Rewrite using right-recursion

```
S → β S'
S' → α S' | ε
```

- Concretely

```
T -> T + id
T-> id
```

- T generates strings
  id
  id+id
  id+id+id …
- Rewrite using right-recursion

```
T -> id
T -> id T
```

More Elimination of Left-Recursion

- In general

  $S \rightarrow S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$
- All strings derived from S start with one of $\beta_1,\ldots,\beta_m$ and continue with several instances of $\alpha_1,\ldots,\alpha_n$
- Rewrite as

  $S \rightarrow \beta_1\ S' \mid \ldots \mid \beta_m\ S'$

  $S' \rightarrow \alpha_1\ S' \mid \ldots \mid \alpha_n\ S' \mid \varepsilon$

General Left Recursion

- The grammar

  $S \rightarrow A\ \alpha \mid \delta$

  $A \rightarrow S\ \beta$

  is also left-recursive because

  $S \rightarrow^{+} S\ \beta\ \alpha$

  where $\rightarrow^{+}$ means "can be rewritten in one or more steps"
- This indirect left-recursion can also be automatically eliminated

Summary of Recursive Descent

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically
- Unpopular because of backtracking
  - Thought to be too inefficient
- In practice, backtracking is eliminated by further restricting the grammar to allow us to successfully *predict* which rule to use

Predictive Parsers

- That there can be many rules for a non-terminal makes parsing hard
- A *predictive parser* processes the input stream typically from left to right
  - Is there any other way to do it? Yes for programming languages!
- It uses information from peeking ahead at the *upcoming terminal symbols* to decide which grammar rule to use next
- And *always* makes the right choice of which rule to use
- How much it can peek ahead is an issue

Predictive Parsers

- An important class of predictive parser only peek ahead one token into the stream
- An an *LL(k)* parser, does a **L**eft-to-right parse, a **L**eftmost-derivation, and **k-**symbol lookahead
- Grammars where one can decide which rule to use by examining only the *next* token are **LL(1)**
- LL(1) grammars are widely used in practice
  - The syntax of a PL can usually be adjusted to enable it to be described with an LL(1) grammar

Predictive Parser

Example: consider the grammar

S → **if** *E* **then** *S* **else** *S*
S → **begin** *S L*
S → **print** *E*
L → **end**
L → **;** *S L*
E → num = num

An *S* expression starts either with an IF, BEGIN, or PRINT token, and an *L* expression start with an END or a SEMICOLON token, and an *E* expression has only one production.

Remember…

- Given a grammar and a string in the language defined by the grammar …
- There may be more than one way to *derive* the string leading to the *same parse tree*
  - It depends on the order in which you apply the rules
  - And what parts of the string you choose to rewrite next
- All of the derivations are *valid*
- To simplify the problem and the algorithms, we often focus on one of two simple derivation strategies
  - A *leftmost* derivation
  - A *rightmost* derivation

LL(k) and LR(k) parsers

- Two important parser classes are LL(k) and LR(k)
- The name LL(k) means:
  - L: *Left-to-right* scanning of the input
  - L: Constructing *leftmost derivation*
  - k: max # of input symbols needed to predict parser action
- The name LR(k) means:
  - L: *Left-to-right* scanning of the input
  - R: Constructing *rightmost derivation* in reverse
  - k: max # of input symbols needed to select parser action
- A LR(1) or LL(1) parser never need to *"look ahead"* more than *one* input token to know what parser production rule applies

Predictive Parsing and Left Factoring

- Consider the grammar

```
E → T + E
E → T
T → int
T → int * T
T → ( E )
```

Even left recursion is removed, a grammar may not be parsable with a LL(1) parser

- Hard to predict because
  - For T, two productions start with *int*
  - For E, it is not clear how to predict which rule to use
- Must **left-factored** grammar before use for predictive parsing
- Left-factoring involves rewriting rules so that, if a non-terminal has > 1 rule, each begins with a **terminal**

Left-Factoring Example

Add new non-terminals X and Y to factor out **common prefixes** of rules

```
E → T + E
E → T
T → int
T → int * T
T → ( E )
```

⇒

```
E → T X
X → + E
X → ε
T → ( E )
T → int Y
Y → * T
Y → ε
```

For each non-terminal the revised grammar, there is either only one rule or every rule begins with a terminal or ε

Left Factoring

- Consider a rule of the form
  A => a B1 | a B2 | a B3 | … a Bn
- A top down parser generated from this grammar is not efficient as it requires backtracking.
- To avoid this problem we left factor the grammar.
  - Collect all productions with the same left hand side and begin with the same symbols on the right hand side
  - Combine common strings into a single production and append a new non-terminal to end of this new production
  - Create new productions using this new non-terminal for each of the suffixes to the common production.
- After left factoring the above grammar is transformed into:
  A –> a A1
  A1 -> B1 | B2 | B3 … Bn

Using Parsing Tables

- LL(1) means that for each non-terminal and token there is only **one** production
- Can be represented as a simple table
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one rule's action or empty if error
- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token *a*
  - And chose the production shown at table cell [S, a]
- Use a stack to keep track of pending non-terminals
- Reject when we encounter an error state, accept when we encounter end-of-input

LL(1) Parsing Table Example

Left-factored grammar

```
E → T X
X → + E | ε
T → ( E ) | int Y
Y → * T | ε
```

End of input symbol

The LL(1) parsing table

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Table Example


```
E → T X
X → + E | ε
T → ( E ) | int Y
Y → * T | ε
```


- Consider the **[E, int]** entry
  - "When current non-terminal is E & next input *int*, use production E→ T X"
  - It's the only production that can generate an *int* in next place
- Consider the **[Y, +]** entry
  - "When current non-terminal is Y and current token is +, get rid of Y"
  - Y can be followed by + only in a derivation where Y→ε
- Consider the **[E, *]** entry
  - Blank entries indicate error situations
  - "There is no way to derive a string starting with * from non-terminal E"

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

LL(1) Parsing Algorithm

```
initialize stack = <S $> and next
repeat
   case stack of
      <X, rest>  : if T[X,*next] = Y1…Yn
                          then stack ← <Y1… Yn rest>;
                          else  error ();
      <t, rest>   : if t == *next ++
                          then  stack ← <rest>;
                          else error ();
until stack == < >
```

where:
(1) next points to the next input token
(2) X matches some non-terminal
(3) t matches some terminal

LL(1) Parsing Example

Stack	Input	Action
`E $`	`int * int $`	`pop();push(T X)`
`T X $`	`int * int $`	`pop();push(int Y)`
`int Y X $`	`int * int $`	`pop();next++`
`Y X $`	`* int $`	`pop();push(* T)`
`* T X $`	`* int $`	`pop();next++`
`T X $`	`int $`	`pop();push(int Y)`
`int Y X $`	`int $`	`pop();next++;`
`Y X $`	`$`	`pop()`
`X $`	`$`	`pop()`
`$`	`$`	**ACCEPT!**

```
E → TX
X → +E
X → ε
T → (E)
T → int Y
Y → *T
Y → ε
```


	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

Constructing Parsing Tables

- No table entry can be multiply defined
- If $A \to \alpha$, where in the line of A we place α ?
- In column t where t can start a string derived from α
  - $\alpha \to^* t\ \beta$
  - We say that $t \in \text{First}(\alpha)$
- In the column t if α is ε and t can follow an A
  - $S \to^* \beta\ A\ t\ \delta$
  - We say $t \in \text{Follow}(A)$

Computing First Sets

Definition: $\text{First}(X) = \{t \mid X \to^* t\alpha\} \cup \{\varepsilon \mid X \to^* \varepsilon\}$

Algorithm sketch (see book for details):

1. for all terminals t do $\text{First}(t) \leftarrow \{ t \}$
2. for each production $X \to \varepsilon$ do $\text{First}(X) \leftarrow \{ \varepsilon \}$
3. if $X \to A_1 \ldots A_n\ \alpha$ and $\varepsilon \in \text{First}(A_i)$, $1 \leq i \leq n$ do add $\text{First}(\alpha)$ to $\text{First}(X)$
4. for each $X \to A_1 \ldots A_n$ s.t. $\varepsilon \in \text{First}(A_i)$, $1 \leq i \leq n$ do add ε to $\text{First}(X)$
5. repeat steps 4 and 5 until no First set can be grown

First Sets. Example

Recall the grammar

$E \to T\ X \qquad\qquad X \to +\ E \mid \varepsilon$

$T \to (\ E\ ) \mid \text{int}\ Y \qquad Y \to *\ T \mid \varepsilon$

First sets

First(( ) = { (}	First(T) = {int, (}
First( )) = {) }	First(E) = {int, (}
First(int) = { int }	First(X) = {+, ε }
First(+) = { + }	First(Y) = {*, ε }
First(*) = { * }	

Computing Follow Sets

- Definition:

  $\text{Follow}(X) = \{ t \mid S \to^* \beta\ X\ t\ \delta \}$

- Intuition
  - If S is the start symbol then $\$ \in \text{Follow}(S)$
  - If $X \to A\ B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and $\text{Follow}(X) \subseteq \text{Follow}(B)$
  - Also if $B \to^* \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$

Computing Follow Sets

Algorithm sketch:

1. Follow(S) ← { \$ }
2. For each production $A \to \alpha\ X\ \beta$
   - add First(β) - {ε} to Follow(X)
3. For each $A \to \alpha\ X\ \beta$ where $\varepsilon \in$ First(β)
   - add Follow(A) to Follow(X)

- repeat step(s) ___ until no Follow set grows

Follow Sets. Example

- Recall the grammar

  $E \to T\ X$ $\quad\quad$ $X \to + E \mid \varepsilon$

  $T \to (E) \mid \text{int}\ Y$ $\quad\quad$ $Y \to * T \mid \varepsilon$

- Follow sets

  Follow(+) = { int, (} $\quad$ Follow(*) = { int, (}

  Follow(( ) = { int, (} $\quad$ Follow(E) = {), \$}

  Follow(X) = {\$,) } $\quad$ Follow(T) = {+,) , \$}

  Follow( )) = {+,) , \$} $\quad$ Follow(Y) = {+,) , \$}

  Follow(int) = {*, +,) , \$}

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G
- For each production $A \to \alpha$ in G do:
  - For each terminal $t \in$ First(α) do
    - T[A, t] = α
  - If $\varepsilon \in$ First(α), for each $t \in$ Follow(A) do
    - T[A, t] = α
  - If $\varepsilon \in$ First(α) and \$ $\in$ Follow(A) do
    - T[A, \$] = α

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1)
- Reasons why a grammar is not LL(1) include
  - G is ambiguous
  - G is left recursive
  - G is not left-factored
- Most programming language grammars are not strictly LL(1)
- There are tools that build LL(1) tables

Bottom-up Parsing

- YACC uses bottom up parsing. There are two important operations that bottom-up parsers use: **shift** and **reduce**
  - In abstract terms, we do a simulation of a Push Down Automata as a finite state automata
- Input: given string to be parsed and the set of productions.
- Goal: Trace a rightmost derivation in reverse by starting with the input string and working backwards to the start symbol

Algorithm

1. Start with an empty stack and a full input buffer. (The string to be parsed is in the input buffer.)
2. Repeat until the input buffer is empty and the stack contains the start symbol.
   a. Shift zero or more input symbols onto the stack from input buffer until a handle (beta) is found on top of the stack. If no handle is found report syntax error and exit.
   b. Reduce handle to the nonterminal A. (There is a production *A -> beta*)
3. Accept input string and return some representation of the derivation sequence found (e.g.., parse tree)

- The four key operations in bottom-up parsing are shift, reduce, accept and error.
- Bottom-up parsing is also referred to as shift-reduce parsing.
- Important thing to note is to know when to shift and when to reduce and to which reduce.

Example of Bottom-up Parsing

STACK	INPUT BUFFER	ACTION
$	num1+num2*num3$	shift
$num1	+num2*num3$	reduc
$F	+num2*num3$	reduc
$T	+num2*num3$	reduc
$E	+num2*num3$	shift
$E+	num2*num3$	shift
$E+num2	*num3$	reduc
$E+F	*num3$	reduc
$E+T	*num3$	shift
E+T*	num3$	shift
E+T*num3	$	reduc
E+T*F	$	reduc
E+T	$	reduc
E	$	accept

```
E -> E+T
   | T
   | E-T
T -> T*F
   | F
   | T/F
F -> (E)
   | id
   | -E
   num
```