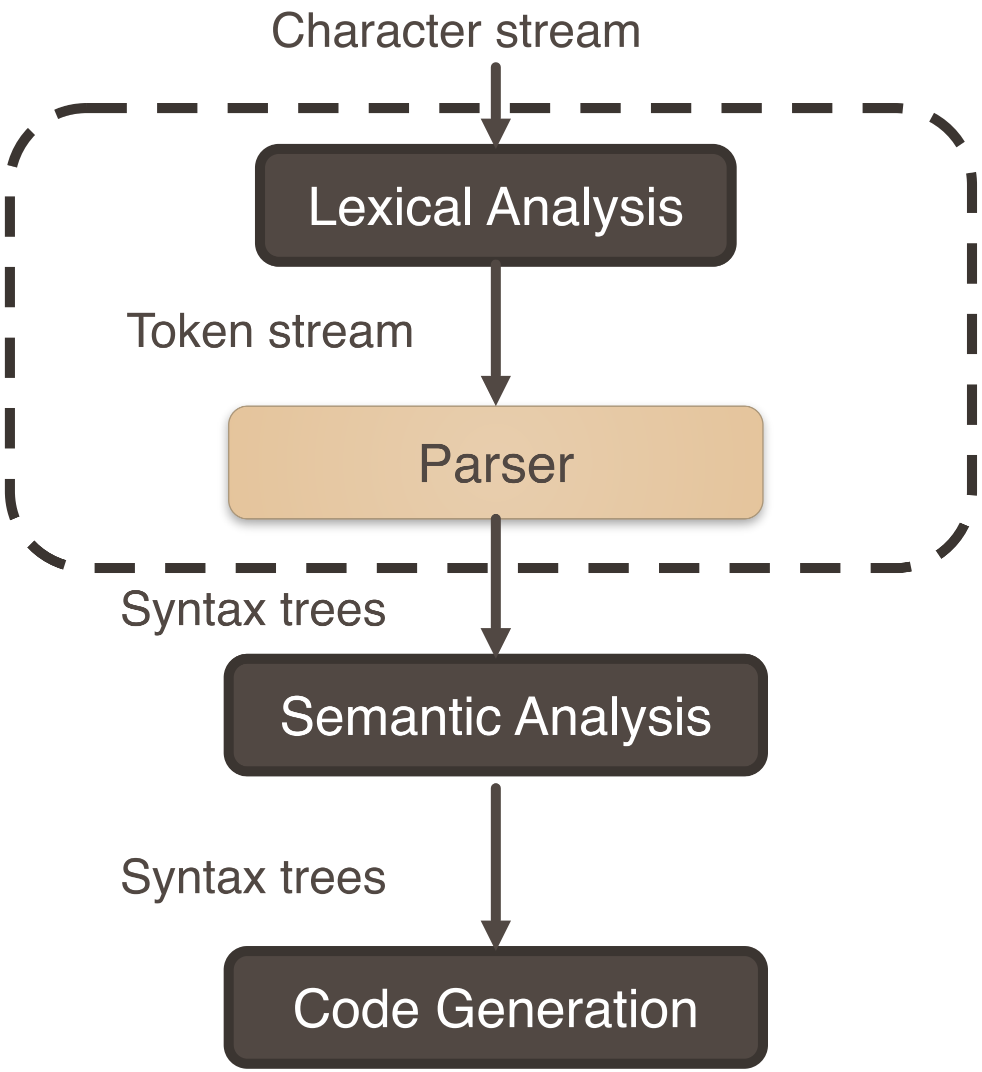
PARSING

Baishakhi Ray

- <id, x> <op, *> <op, %>
    - Is it a valid token stream in C language?
    - Is it a valid statement in C language?

Intro to Parsing



- Not every strings of tokens are valid
- Parser must distinguish between valid and invalid token strings.
- We need
  - A Language: to describe what is valid string?
  - A method: to determine membership of inputs in this language.

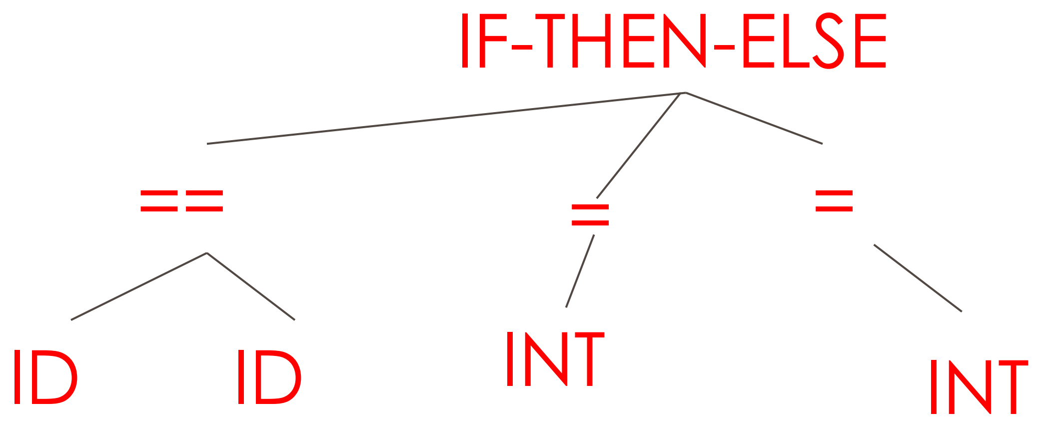
Intro to Parsing

- Input: if(x==y) 1 else 2;

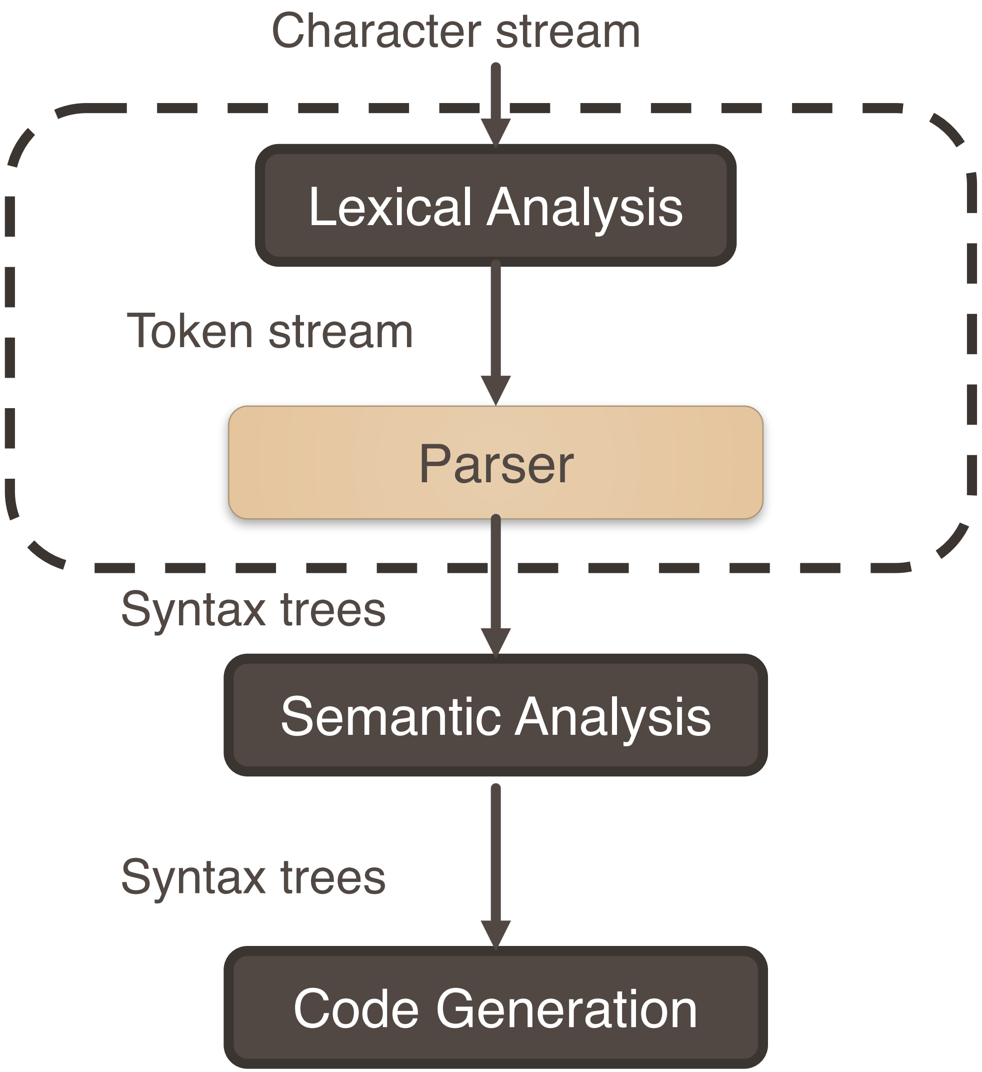
- Parser Input (Lexical Input):

KEY(IF) ‘(‘ ID(x) OP(‘==‘) ‘)’ INT(1) KEY(ELSE) INT(2) ‘;’

- Parser Output



Intro to Parsing



- Not every strings of tokens are valid
- Parser must distinguish between valid and invalid token strings.
- We need
  - A Language: to describe what is valid string?
    - Context Free Grammar
    - Capture Language Syntax
  - A method: to determine membership of inputs in this language.

Context Free Grammar

- A CFG consists of
  - A set of terminal T
  - A set of non-terminal N
  - A start symbol S (S ϵ N)
  - A set of production rules
    - X -> $Y_1 \ldots\ldots Y_N$
    - X ϵ N
    - $Y_i \ \epsilon \ \{N, T, \varepsilon\}$
- Ex: S -> (S) l ε
  - N = {S}
  - T = { (,) , ε}

Context Free Grammar

1. Begin with a string with only the start symbol S
2. Replace a non-terminal X with in the string by the RHS of some production rule:

   $X \text{->} Y_1 \ldots\ldots Y_n$

3. Repeat 2 again and again until there are no non-terminals

$X_1 \ldots\ldots X_i \underline{X} X_{i+1} \ldots\ldots X_n \text{->} X_1 \ldots\ldots X_i Y_1 \ldots\ldots Y_k X_{i+1} \ldots\ldots X_n$

For the production rule $X \text{->} Y_1 \ldots\ldots Y_k$

$$\alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \ldots \rightarrow \alpha_n$$

$$\alpha_0 \xrightarrow{*} \alpha_n, n \geq 0$$

Context Free Grammar

- Let G be a CFG with start symbol S. Then the language L(G) of G is:

$$\{a_1 \ldots a_i \ldots a_n \,|\, \forall i a_i \in T \wedge S \xrightarrow{*} a_1 \ldots a_i \ldots a_n\}$$

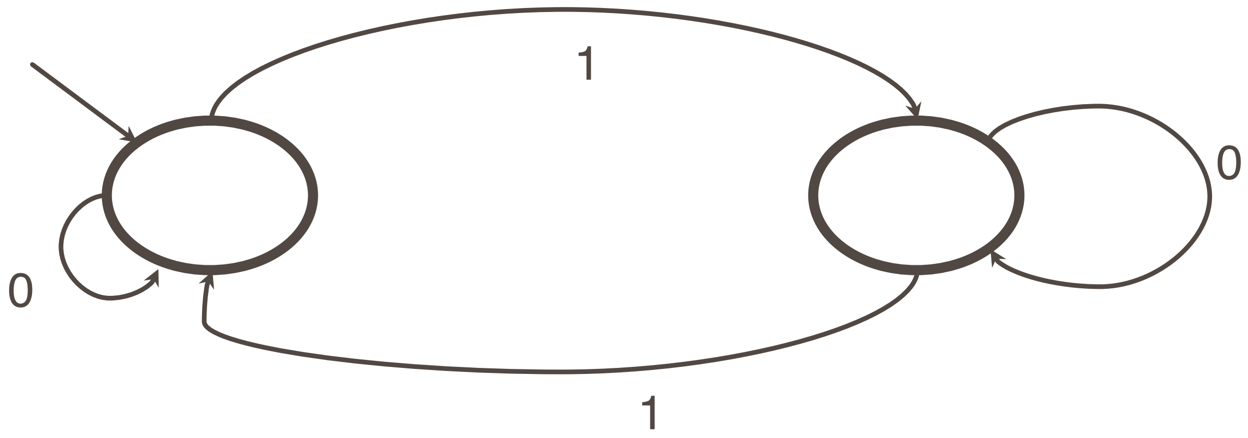
Context Free Grammar

- There are no rules to replace terminals.
- Once generated, terminals are permanent
- Terminals ought to be tokens of programming languages
- Context-free grammars are a natural notation for this recursive structure

Languages and Automata

- Formal languages are very important in programming languages
- Regular Languages
  - Weakest formal languages that are widely used
  - Many applications
- Many Languages are not regular

Automata that accept odd numbers of 1



How many 1s it has accepted?

- Only solution is duplicate state

Automata do not have any memory

Intro to Parsing

- Regular Languages
  - Weakest formal languages that are widely used
  - Many applications
- Consider the language $\{(^i\ )^i \mid i \geq 0\}$
  - (), (( )), ((( )))
  - ((1 + 2) * 3)
- Nesting structures
  - if .. if.. else.. else..

Regular languages cannot handle well

CFG: Simple Arithmetic expression

E → E + E

| E * E

| (E)

| id

Languages can be generated: id, (id), (id + id) * id, …

CFG: Exercise

$$S \rightarrow aXa$$
$$X \rightarrow \varepsilon \,|\, bY$$
$$Y \rightarrow \varepsilon \,|\, cXc$$

Some Valid Strings are: aba, abcca, …

Derivation

- A derivation is a sequence of production
  - S -> … -> … ->

- A derivation can be drawn as a tree
  - Start symbol is tree's root
  - For a production X -> Y_1….Y_n, add children Y_1….Y_n to node X

- Grammar
  - E -> E + E I E * E I (E) I id
- String
  - id * id + id

- Derivation

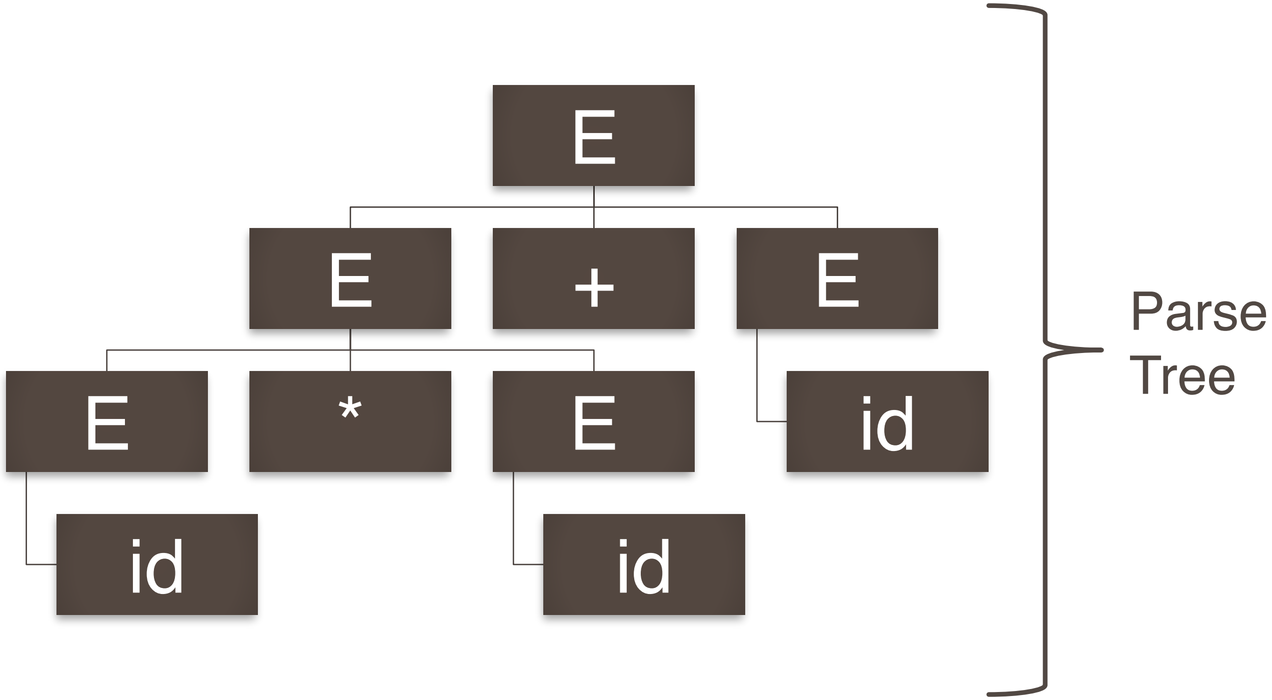
E -> E + E

-> E * E + E

-> id * E + E

-> id * id + E

-> id * id + id



Parse Tree

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes

- An in-order traversal of the leaves is the original input

- The parse tree shows the association of operations, the input string does not

Parse Tree

- Left-most derivation
    - At each step, replace the left-most non-terminal

E -> E + E

-> E * E + E

-> id * E + E

-> id * id + E

-> id * id + id

- Right-most derivation
    - At each step, replace the right-most non-terminal

E -> E + E

-> E + id

-> E * E + id

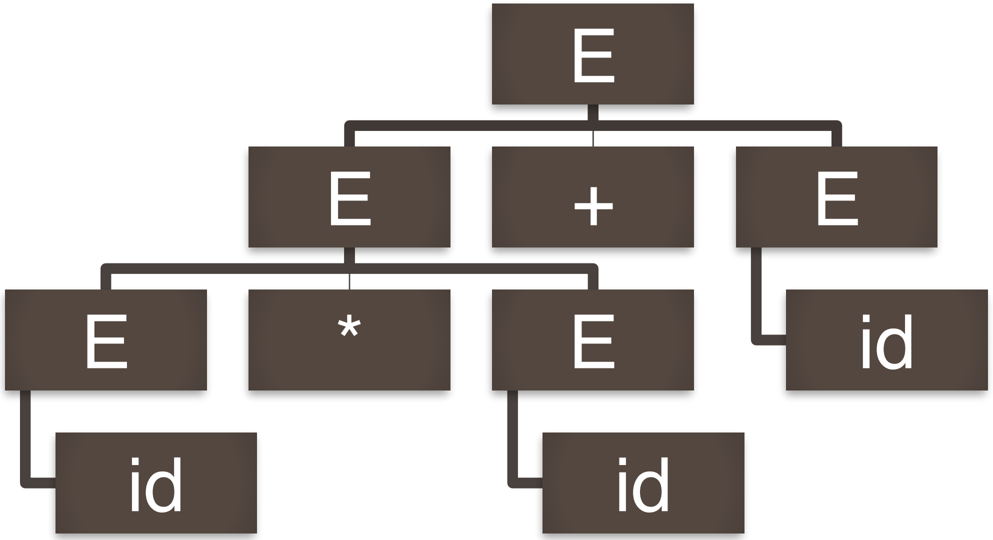
-> E * id + id

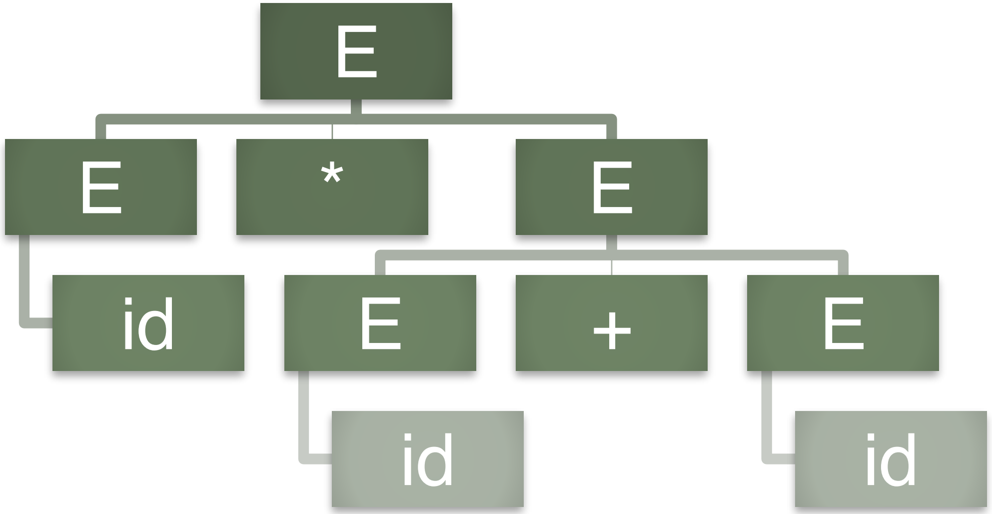
-> id * id + id

Note that, right-most and left-most derivations have the same parse tree

Ambiguity

- Grammar
  - E -> E + E I E * E I (E) I id
- String
  - id * id + id





Ambiguity

- A grammar is ambiguous if it has more than one parse tree for a string
  - There are more than one right-most or left-most derivation for some string

- Ambiguity is bad
  - Leaves meaning for some programs ill-defined

Example of Ambiguous Grammar

- S->SS | a | b

Resolving Ambiguity

- Most direct way to rewrite the grammar unambiguously

$$id * id + id$$

$$E = E' + E \,|\, E'$$
$$E' = id * E' \,|\, id \,|\, (E) * E' \,|\, (E)$$

Resolving Ambiguity

- Impossible to convert ambiguous to unambiguous grammar automatically

- Instead of rewriting
  - Use ambiguous grammar
  - Along with disambiguating rules
    - Eg, precedence and associativity rules
    - Enforces precedence of * over +
    - associativity: %left +

Abstract Syntax Trees

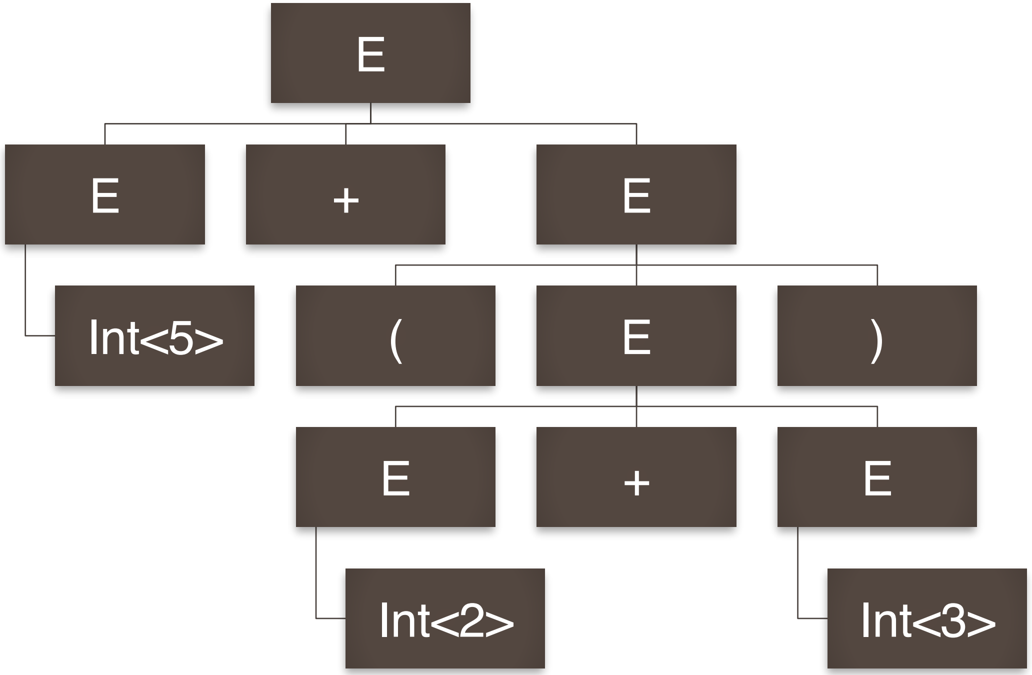
- A parser traces the derivation of a sequence of tokens
- But the rest of the compiler needs a structural representation of the program
- Abstract Syntax Trees
  - Like parse trees but ignore some details
  - Abbreviated as AST

Abstract Syntax Trees

- Grammar
  - E -> int | (E) | E + E
- String
  - 5 + (2 + 3)
- After lexical analysis
  - Int<5> ‘+’ ‘(‘ Int<2> ‘+’ Int<3> ‘)’

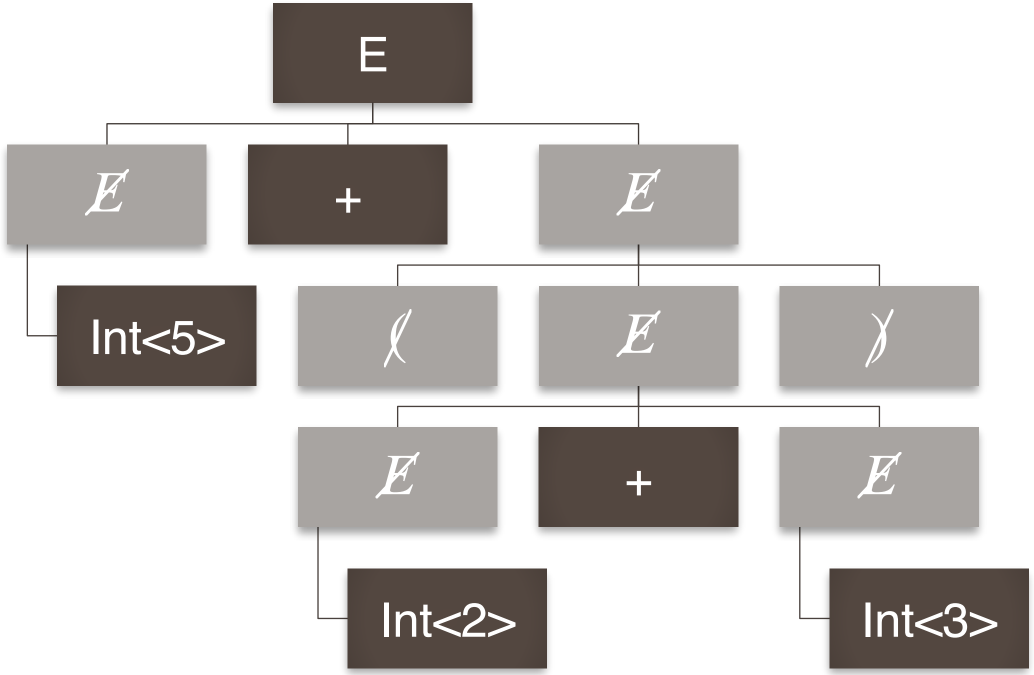
Abstract Syntax Trees: 5 + (2 + 3)

Parse Trees



Abstract Syntax Trees: 5 + (2 + 3)

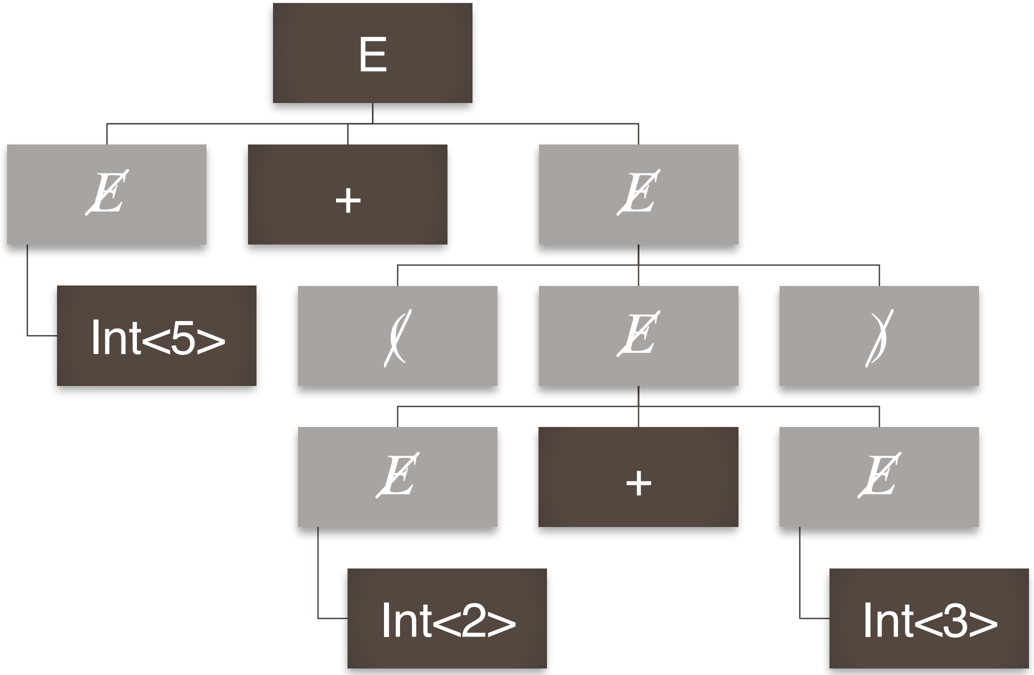
Parse Trees



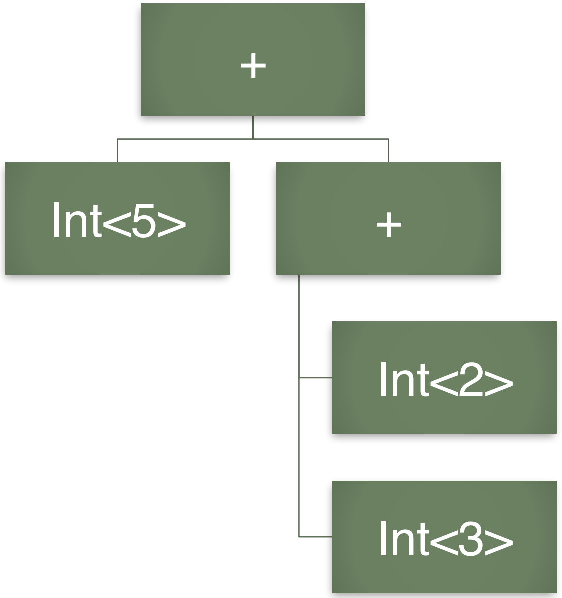
- Have too much information
  - Parentheses
  - Single-successor nodes

Abstract Syntax Trees: 5 + (2 + 3)

Parse Trees



AST



- Have too much information
  - Parentheses
  - Single-successor nodes

- ASTs capture the nesting structure
- But abstracts from the concrete syntax
  - More compact and easier to use

Error Handling

- Purpose of the compiler is
  - To detect non-valid programs
  - To translate the valid ones
- Many kinds of possible errors (e.g., in C)

Error Kind		Example	Detected by
Lexical	Misspelling of identifiers, keywords, or operators.	… $ ...	Lexer
Syntax	Misplaced operators, semicolons, braces, switch-case statements, etc.	… x*%...	Parser
Semantic	Type mismatches between operators and operands	… int x; y = x(3);...	Type Checker
Correctness	Incorrect reasoning	Using = instead of ==	tester/user

Error Handling

- Error Handler should
  - Discover errors accurately and quickly
  - Recover from an error quickly
  - Not slow down compilation of valid code

- Types of Error Handling
  - Panic mode
  - Error productions
  - Automatic local or global correction

Panic Mode Error Handling

- Panic mode is simplest and most popular method

- When an error is detected
  - Discard tokens until one with a clear role is found
    - Typically looks for “synchronizing” tokens
      - Typically the statement of expression terminators
      - Example: delimiters (; }, etc.)
  - Continue from there

Panic Mode Error Handling

- Example:
  - (1 + + 2) + 3
- Panic-mode recovery:
  - Skip ahead to the next integer and then continue
- Bison: use the special terminal error to describe how much input to skip
  - E -> int | E + E | (E) | error int | (error)

Normal mode   Error mode

Error Productions

- Specify known common mistakes in the grammar

- Example:
  - Write 5x instead of 5 * x
  - Add production rule E -> .. | E E

- Disadvantages
  - complicates the grammar

Error Corrections

- Idea: find a correct “nearby” program
  - Try token insertions and deletions (goal: minimize edit distance)
  - Exhaustive search

- Disadvantages
  - Hard to implement
  - Slows down parsing of correct programs
  - “Nearby” is not necessarily “the intended” program

Error Corrections

- Past
  - Slow recompilation cycle (even once a day)
  - Find as many errors in one cycle as possible

- Disadvantages
  - Quick recompilation cycle
  - Users tend to correct one error/cycle
  - Complex error recovery is less compelling

Parsing algorithm: Recursive Descent Parsing

- The parse tree is constructed
  - From the top
  - From left to right

- Terminals are seen in order of appearance in the token stream

Parsing algorithm: Recursive Descent Parsing

- Grammar:
  - E -> T | T + E
  - T -> int | int * T | (E)

- Token Stream: (int<5>)

- Start with top level non-terminal E
  - Try the rules for E in order

Recursive Descent Parsing Example

E -> T | T + E

T -> int | int * T | (E)

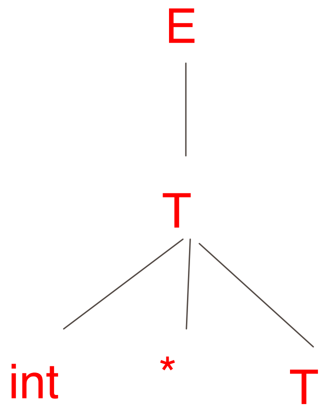
mismatch: int does not match arrowhead (
backtrack

(int<5>)

Recursive Descent Parsing Example

E -> T | T + E

T -> int | int * T | (E)



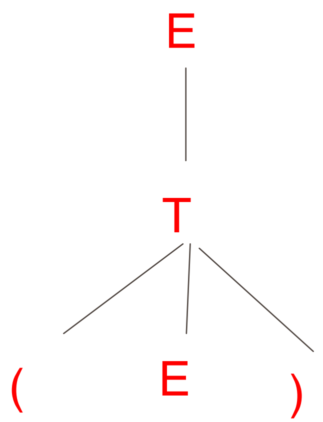
backtrack

(int<5>)

Recursive Descent Parsing Example

E -> T | T + E

T -> int | int * T | (E)



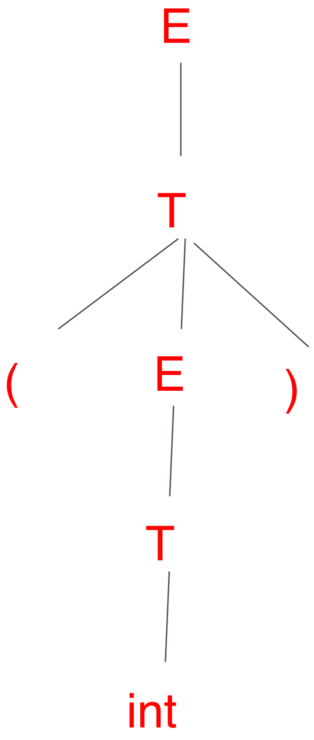
Match! Advance input

(int<5>)

Recursive Descent Parsing Example

E -> T | T + E

T -> int | int * T | (E)



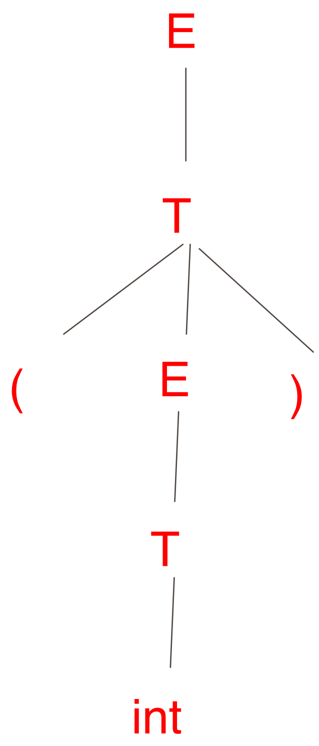
Match! Advance input

(int<5>)

Recursive Descent Parsing Example

E -> T | T + E

T -> int | int * T | (E)



Match! Advance input

(int<5>)

$$E \rightarrow E' \mid E' + E$$

$$E' \rightarrow -E' \mid id \mid (E)$$

Input: id + id

A Recursive Descent Parser. Preliminaries

- TOKEN: type of tokens
  - Special tokens INT, OPEN, CLOSE, PLUS, TIMES

- The global next point to the next token

A Top Down Parsing Algorithm

```
void A() {
  Choose an A-production: A -> S_1 S_2 ... S_k;
  for (i=1 or k) {
     if (S_i is a nonterminal)
         Call S_i();
     else if (X_i == current input TOKEN tok). /*terminal*/
            next++;
  }
}
```

Recursion without backtracking

A (Limited) Recursive Descent Parser

- Define boolean functions that check the token string for a match of
  - A given token terminal

    ```
    bool term (TOKEN tok) { return *next++ == tok; }
    ```

  - The n^{th} production of S:

    ```
    bool S_n() { … }
    ```

  - Try all productions of S:

    ```
    bool S() { … }
    ```

A (Limited) Recursive Descent Parser

- For production E → T

```
bool E1() { return T(); }
```

- For production E → T + E

```
bool E2() { return T() && term(PLUS) && E(); }
```

- For all productions of E (with backtracking)

```
bool E() {
  TOKEN *save = next;
  return (next = save, E1( )) || (next = save, E2( ));
}
```

Grammar:

E -> T | T + E

T -> int | int * T | (E)

A (Limited) Recursive Descent Parser (4)

- Functions for non-terminal T

```
bool T1() { return term(INT); }

bool T2() { return term(INT) && term(TIMES) && T(); }

bool T3() { return term(OPEN) && E() && term(CLOSE); }


bool T() {
        TOKEN *save = next;
        return (next = save, T1())   || (next = save, T2())   || (next = save, T3());
}
```

Grammar:

E -> T | T + E

T -> int | int * T | (E)

Recursive Descent Parsing

- To start the parser
  - Initialize next to point to first token
  - Invoke E() (start symbol)

Example

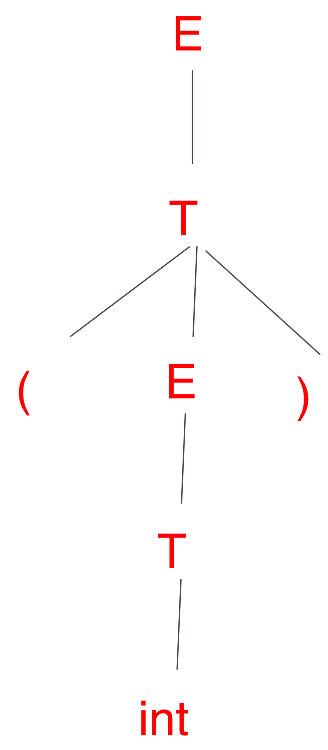
Grammar:
E → T | T + E
T → int | int * T | (E)

Input: (int)

Code:
```
bool term(TOKEN tok) { return *next++ == tok; }

bool E_1() { return T(); }
bool E_2() { return T() && term(PLUS) && E(); }
bool E() {TOKEN *save = next;
        return (next = save, E_1()) || (next = save, E_2()); }

bool T_1() { return term(INT); }
bool T_2() { return term(INT) && term(TIMES) && T(); }
bool T_3() { return term(OPEN) && E() && term(CLOSE); }
bool T() { TOKEN *save = next;
      return (next = save, T_1())
          || (next = save, T_2())
          || (next = save, T_3()); }
```



Example

Grammar:

E → T | T + E

T → int | int * T | (E)

Input: int

Code:

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }
bool E() {TOKEN *save = next;
          return (next = save, E1()) || (next = save, E2()); }

bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }
bool T() { TOKEN *save = next;
      return (next = save, T1())
          || (next = save, T2())
          || (next = save, T3()); }
```

When Recursive Descent Does Not Work

Grammar:

E → T | T + E

T → int | int * T | (E)

Input: int * int

Code:

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }
bool E() {TOKEN *save = next;
          return (next = save, E1()) || (next = save, E2()); }

bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }
bool T() { TOKEN *save = next;
     return (next = save, T1())
         || (next = save, T2())
         || (next = save, T3()); }
```

Recursive Descent Parsing: Limitation

- If production for non-terminal X succeeds
  - Cannot backtrack to try different production for X later
- General recursive descent algorithms support such full backtracking
  - Can implement any grammar
- Presented RDA is not general
  - But easy to implement
- Sufficient for grammars where for any non-terminal at most one production can succeed
- The grammar can be rewritten to work with the presented algorithm
  - By left factoring

Left Factoring

A -> $\alpha\beta 1$ I $\alpha\beta 2$

- The input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta 1$ or $\alpha\beta 2$.
- We can defer the decision by expanding A to αA'.
- Then, after seeing the input derived from α, we expand A' to $\beta 1$ or $\beta 2$ (left-factored)
- The original productions become:

A -> $\alpha A'$, A' -> $\beta 1$ I $\beta 2$

Left Factoring

- Recall the grammar

  E → T + E | T

  T → int | int * T | (E)

- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict
- We need to left-factor the grammar

Left-Factoring Example

- Grammar

  E → T + E | T

  T → int | int * T | (E)

- Factor out common prefixes of productions

  E → T X

  X → + E | ε

  T → (E) | int Y

  Y → * T | ε

When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S\ a$

```
bool S1() { return S() && term(a); }
bool S() { return S1(); }
```

- S() goes into an infinite loop
- A left-recursive grammar has a non-terminal S

  $S \rightarrow^{+} S\alpha$ for some α

- Recursive descent does not work for left recursive grammar

Elimination of Left Recursion

- Consider the left-recursive grammar

$$S \rightarrow S\ \alpha \mid \beta$$

- S generates all strings starting with a β and followed by a number of α
- Can rewrite using right-recursion

$$S \rightarrow \beta\ S'$$

$$S' \rightarrow \alpha\ S' \mid \varepsilon$$

More Elimination of Left-Recursion

- In general

  $S \to S\ \alpha_1 \mid \ldots \mid S\ \alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$

- 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

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

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

General Left Recursion

- The grammar

  $S \to A\,\alpha \mid \delta$

  $A \to S\,\beta$

  is also left-recursive because

  $S \to^{+} S\,\beta\,\alpha$

- This left-recursion can also be eliminated

Example

- S-> Aa | b
- A —> A c | S d | ϵ
- Remove Recursion.

- S - > A a l b.
- A -> b d A’ l A’
- A’ -> cA’ l a d A’ l a l ϵ

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 restricting the grammar

Predictive Parsers

- Like recursive-descent but parser can "predict" which production to use
  - By looking at the next few tokens
  - No backtracking

- Predictive parsers accept LL(k) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - k means "predict based on k tokens of lookahead"
  - In practice, LL(1) is used

LL(1) vs. Recursive Descent

- In recursive-descent
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices
- In LL(1)
  - At each step, only one choice of production
  - That is
    - When a non-terminal A is leftmost in a derivation
    - The next input symbol is t
    - There is a unique production $A \to \alpha$ to use
      - Or no production to use (an error state)
- LL(1) is a recursive descent variant without backtracking

Predictive Parsing and Left Factoring

- Recall the grammar

  E → T + E | T

  T → int | int * T | (E)

- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict
- We need to left-factor the grammar

Left-Factoring Example

- Grammar

  E → T + E | T

  T → int | int * T | (E)

- Factor out common prefixes of productions

  E → T X

  X → + E | ε

  T → (E) | int Y

  Y → * T | ε

LL(1) Parsing Table Example

- Left-factored grammar

  E → T X

  X → + E | ε

  T → (E) | int Y

  Y → * T | ε

- The LL(1) parsing table:

Left-most non-terminals		next input tokens					
		int	*	+	(	)	$
	E	TX			TX		
	X			+E		ε	ε
	T	int Y			(E)		
	Y		*T	ε		ε	ε

LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - “When current non-terminal is E and next input is int, use production E → T X”
  - This can generate an int in the first position
- Consider the [Y,+] entry
  - “When current non-terminal is Y and current token is +, get rid of Y”
  - Y can be followed by + only if Y → ε

LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
- Consider the [E,*] entry
  - "There is no way to derive a string starting with * from non-terminal E"

Using Parsing Tables

- Method similar to recursive descent, except
  - For the leftmost non-terminal S
  - We look at the next input token a
  - And choose the production shown at [S,a]
- Reject on reaching error state
- Accept on end of input & empty stack

Bottom-Up Parsing

- Bottom-up parsing is more general than (deterministic) top-down parsing
  - just as efficient
  - Builds on ideas in top-down parsing
- Bottom-up parsers don't need left-factored grammars
- Revert to the "natural" grammar for our example:

  E → T + E | T

  T → int * T | int | (E) •
- Consider the string: int * int + int

Bottom-Up Parsing

- Revert to the "natural" grammar for our example:

  E → T + E | T

  T → int * T | int | (E) •

- Consider the string: int * int + int

- Bottom-up parsing reduces a string to the start symbol by inverting productions:

`int * int + int`	`T → int`
`int * T + int`	`T → int * T`
`T + int`	`T → int`
`T + T`	`E → T`
`T + E`	`E → T + E`
`E`	

Observation

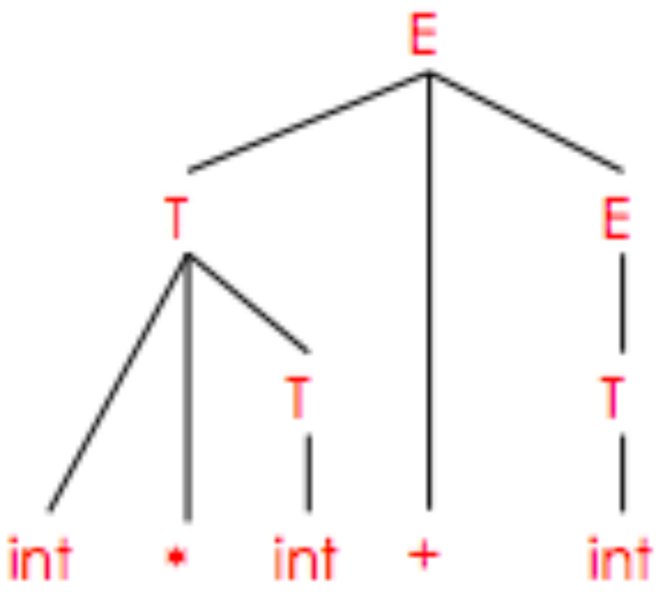
- Read the productions in reverse (from bottom to top)
- This is a rightmost derivation!

`int * int + int`	`T → int`
`int * T + int`	`T → int * T`
`T + int`	`T → int`
`T + T`	`E → T`
`T + E`	`E → T + E`
`E`	

Bottom-Up Parsing

- A bottom-up parser traces a rightmost derivation in reverse

```
int * int + int        T → int
int * T   + int        T → int * T
T + int                T → int
T + T                  E → T
T + E                  E → T + E
E
```



L, R, and all that

- LR parser: “Bottom-up parser”
- L = Left-to-right scan, R = Rightmost derivation
- RR parser: R = Right-to-left scan (from end)
  - nobody uses these
- LL parser: “Top-down parser”:
- L = Left-to-right scan: L = Leftmost derivation
- LR(1): LR parser that considers next token (lookahead of 1)
- LR(0): Only considers stack to decide shift/reduce
- SLR(1): Simple LR: lookahead from first/follow rules Derived from LR(0) automaton
- LALR(1): Lookahead LR(1): fancier lookahead analysis Uses same LR(0) automaton as SLR(1)