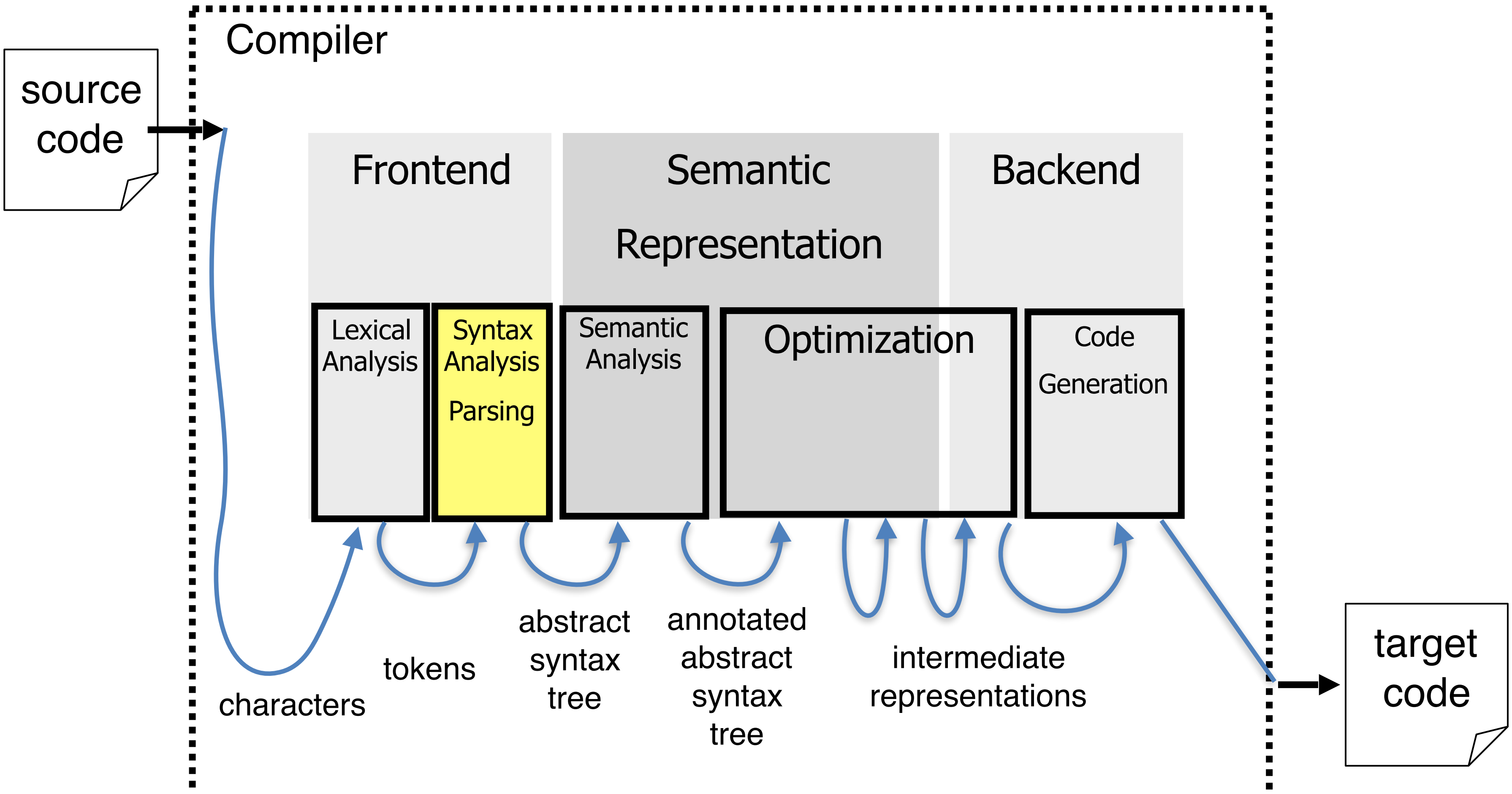
Syntactic Analysis

Greta Yorsh

Anatomy of a modern compiler

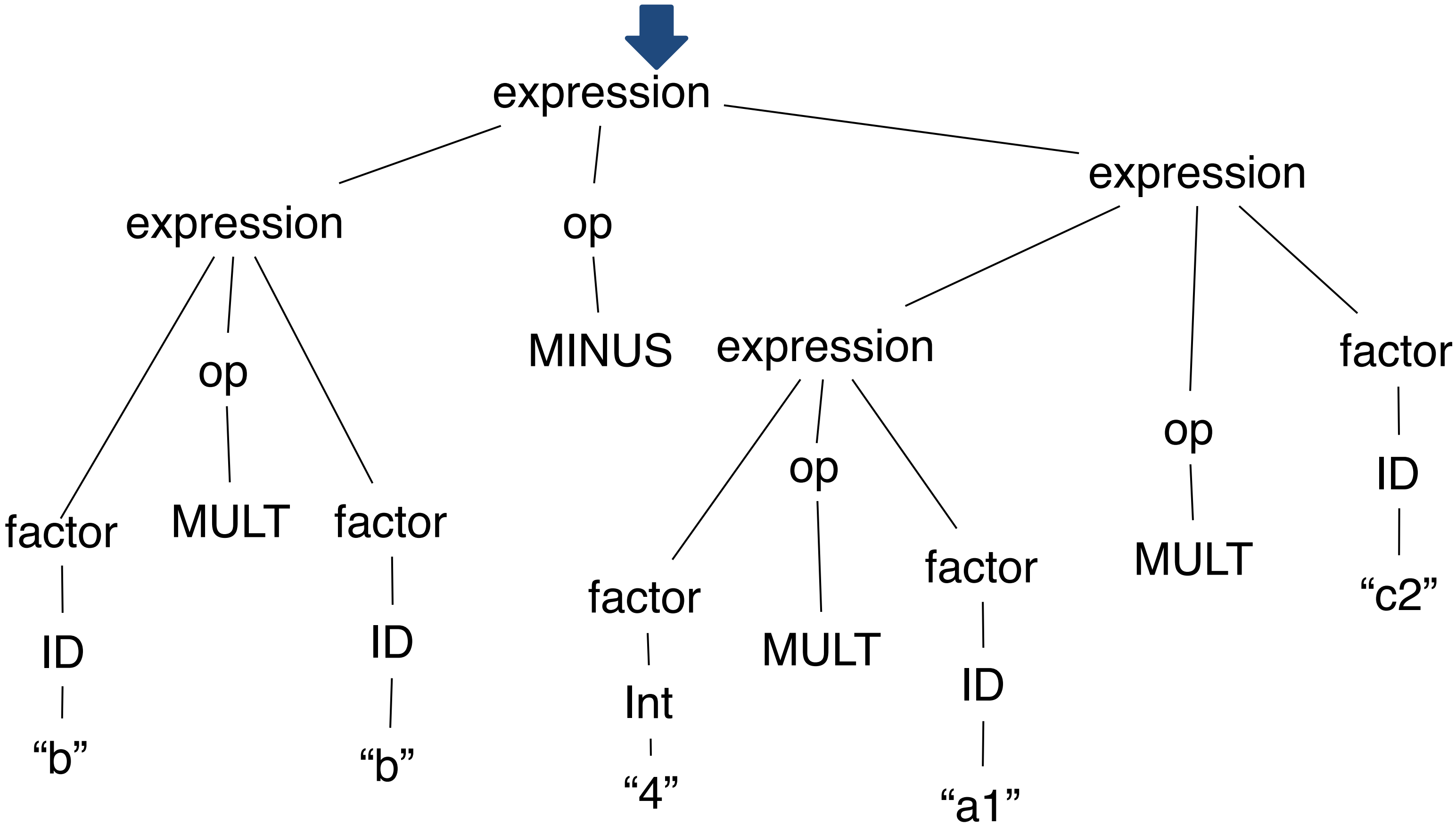


Introduction to Parsing

- Context free grammars
- Ambiguity
- Leftmost and rightmost derivations
- Overview: top-down vs bottom-up parsing

Syntax Tree (Parse Tree)

ID,"x" EQ **ID,"b"** MULT **ID,"b"** MINUS INT,4 MULT **ID,"a1"** MULT **ID,"c2"**



x = b*b – 4*a1*c2 → Lexical Analysis | Syntax Analysis | Semantic Analysis | Optimization | Code Generation

Parsing

- Tasks
  - Check that the sequence of tokens is
    a **well-formed program** in the language
  - Construct a **structured representation** of the input text
  - Error detection and reporting

- Challenges
  - How do you **describe** the programming language?
  - How do you **check** validity of an input?
  - Where do you **report** an error?

Context free grammars

$$G = (V,T,P,S)$$

- V non-terminals (syntactic variables)
- T terminals (tokens)
- P derivation rules (productions)
  - each rule of the form $V \rightarrow (T \mid V)^*$
- S – start symbol

Quick quiz

- Why do we need context free grammars?

Example: matching parenthesis

$S \rightarrow SS$

$S \rightarrow (S)$

$S \rightarrow ()$

Example: arithmetic expressions

$$S \to S\,;\,S$$

$$S \to \mathtt{id}\ :=\ E$$

$$E \to \mathtt{id}\ \mid\ E + E\ \mid\ E * E\ \mid\ (\ E\ )$$

V = { S, E }
T = { id, +, *, (,),:=, ;}

Example: derivation

input

x := z;
y := x + z

grammar

S → S;S
S → id := E
E → id | E + E | E * E | (E)

Derivation	Rule
S	S → S;S
S ; S	S → id := E
id := E ; S	E → id
id := id ; S	S → id := E
id := id ; id := E	E → E + E
id := id ; id := E + E	E → id
id := id ; id := E + id	E → id
id := id ; id := id + id	

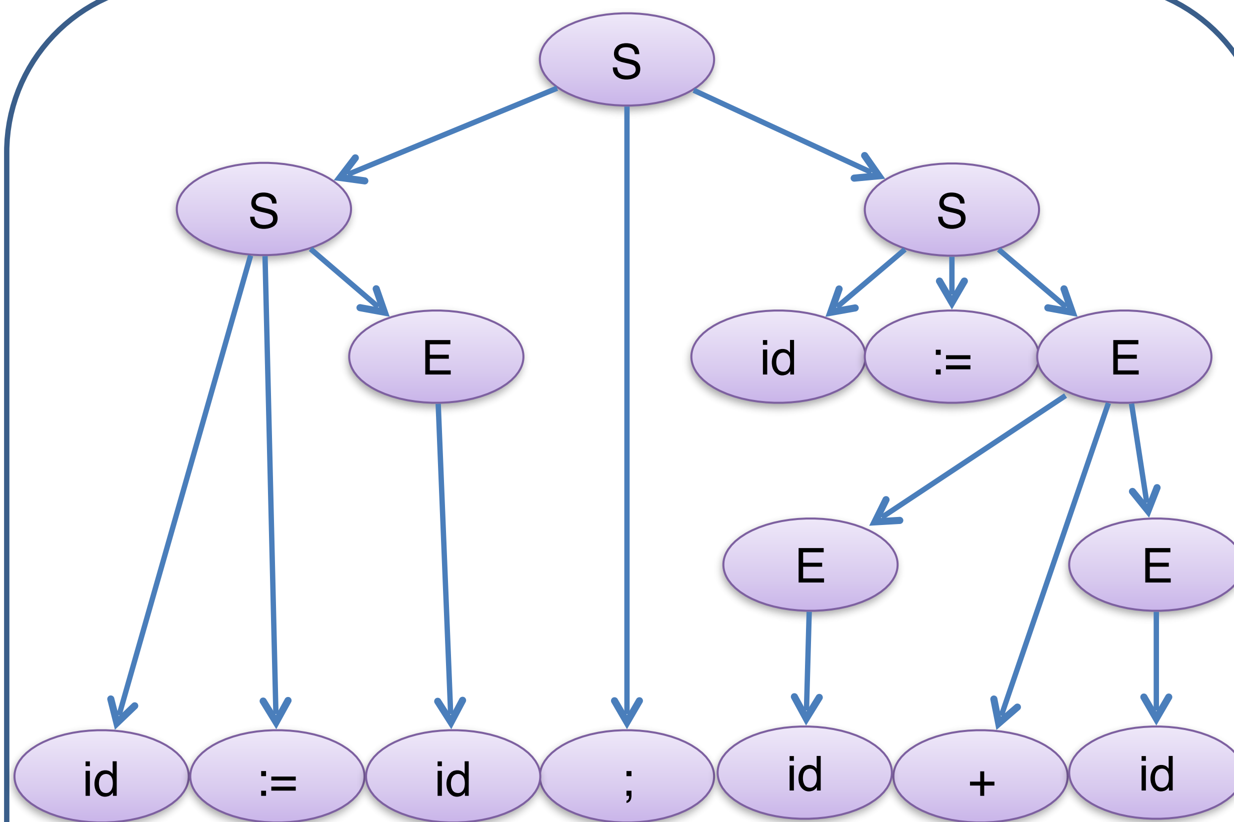
<id,"x"><ASS><id,"z"><SEMI><id,"y"><ASS><id,"x"><PLUS><id,"z">

Terminology

- **Derivation:** a sequence of replacements of non-terminals using the derivation rules
- **Language:** the set of strings of terminals derivable from the start symbol
- **Sentential form:** the result of a partial derivation in which there may be non-terminals

Example: parse tree

```
        S
    S     ;  S
id := E ;   S
id := id;  S
id := id; id := E
id := id; id := E + E
id := id; id := E + id
id := id; id := id + id
x:= z;  y := x  + z
```



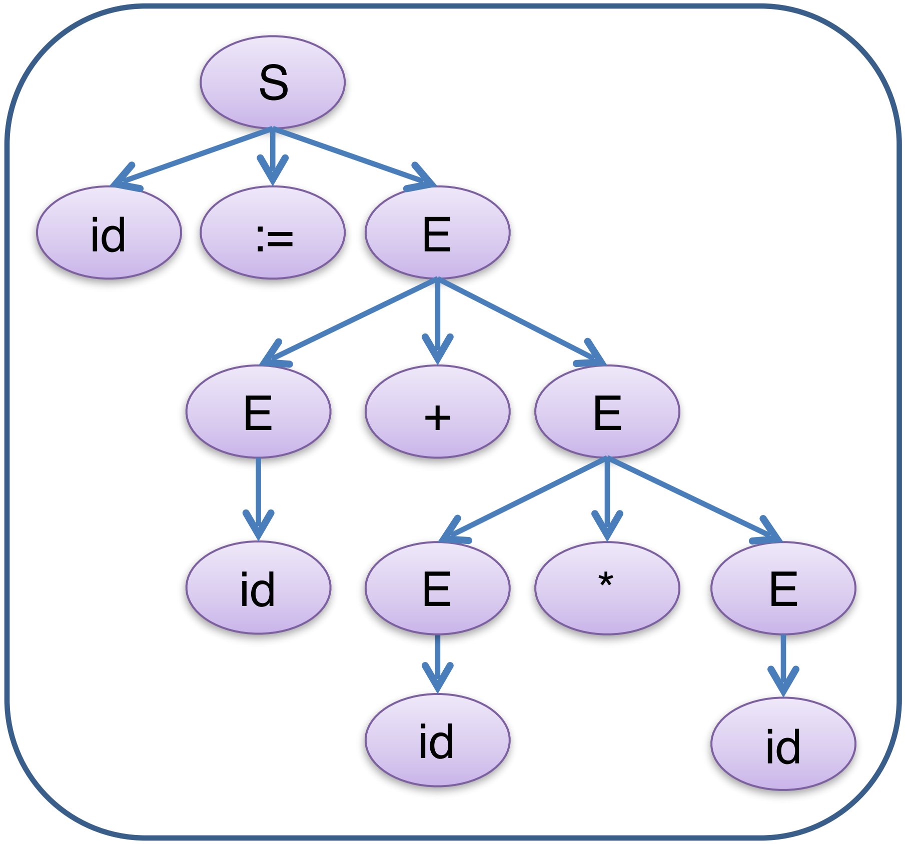
Questions

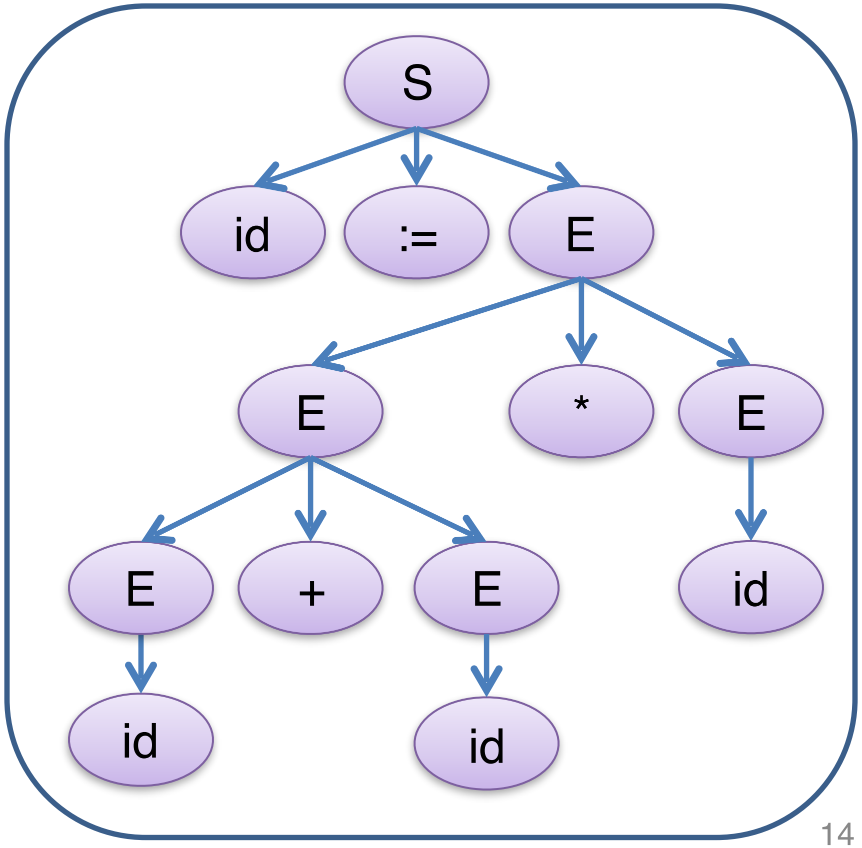
- How did we know which rule to apply on every step?
- Does it matter?
- Would we always get the same result?

Example: ambiguity

x := y+z*w

```
S → S;S
S → id := E
E → id | E + E | E * E | ( E )
```





Derivations: leftmost and rightmost

- **Leftmost** derivation: expand leftmost non-terminal
- **Rightmost** derivation: expand rightmost non-terminal

- Describe derivation by listing the sequence of rules
- Always know what a rule is applied to
- Used in our parsers

Example: **leftmost** derivation

```
x := z;
y := x + z
```

```
S → S;S
S → id := E
E → id | E + E | E * E | ( E )
```

Derivation	Rule
S	S → S;S
S ; S	S → id := E
id := E ; S	E → id
id := id ; S	S → id := E
id := id ; id := E	E → E + E
id := id ; id := E + E	E → id
id := id ; id := id + E	E → id
id := id ; id := id + id	
x:= z ; y := x + z	

Example: **rightmost** derivation

x := z;
y := x + z

```
S → S;S
S → id := E
E → id | E + E | E * E | ( E )
```

Derivation	Rule
S	S → S;S
S ; S	S → id := E
S ; id := E	E → E + E
S ; id := E + E	E → id
S ; id := E + id	E → id
S ; id := id + id	S → id := E
id := E ; id := id + id	E → id
id := id ; id := id + id	
x:= z ; y := x + z	

Example: bottom-up

```
x := z;
y := x + z
```

```
S → S;S
S → id := E
E → id | E + E | E * E | ( E )
```

Bottom-up picking left alternative on every step == Rightmost derivation when going top-down

```
            S                          S → S;S
        S   ;   S                      S → id := E
        S   ;   id := E                E → E + E
        S   ;   id := E + E            E → id
        S   ;   id := E + id           E → id
        S   ;   id := id + id          S → id := E
id := E     ;   id := id + id          E → id
id := id    ;   id := id + id
   x:= z    ;    y := x  + z
```

Parsing

- **Find a derivation from the start symbol to the input word**

- Search problem
- Easy to solve with backtracking
- ...but very expensive

Brute-force parsing

x := z;
y := x + z

```
S → S;S
S → id := E
E → id | E + E | E * E | ( E )
```

```
id := id ; id := id + id
```

E → id

E → id

```
id := E; id := id + id      id := id ;id := E+ id      ...
```

```
id := E; id := id + id      id := E; id := id + id
```

not a parse tree… a search for the parse tree by exhaustively applying all rules

Parsing with pushdown automata

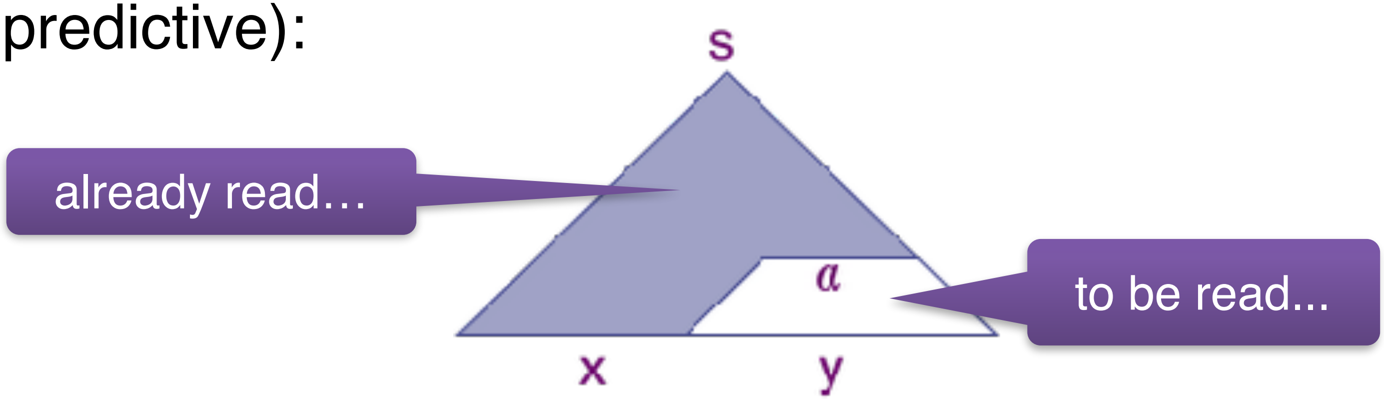
- A context free language can be recognized by **non-deterministic** pushdown automaton
- Cocke-Younger-Kasami (CYK) parser can be used to parse **any** context-free language but has complexity $O(n^3)$
- We want **efficient** parsers
  - linear in input size
  - **deterministic** pushdown automata
  - **sacrifice generality for efficiency**

Efficient Parsers

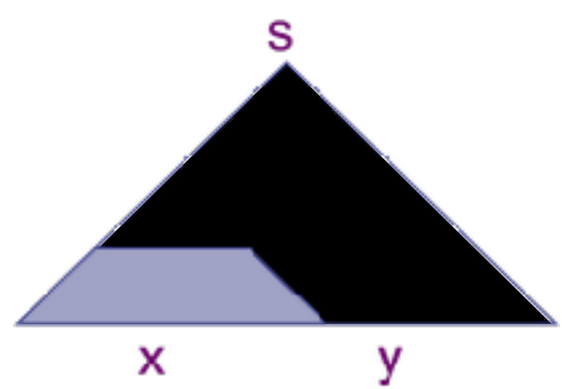
- Top-down (predictive): LL
  - read input from left to right (L)
  - construct the leftmost derivation (L)
  - apply rules “from left to right”
  - predict what rule to apply based on non-terminal and token
- Bottom up (shift-reduce): LR
  - read input from left to right (L)
  - construct the rightmost derivation (R)
  - apply rules “from right to left”
  - reduce a right-hand side of a production to its non-terminal

Efficient Parsers

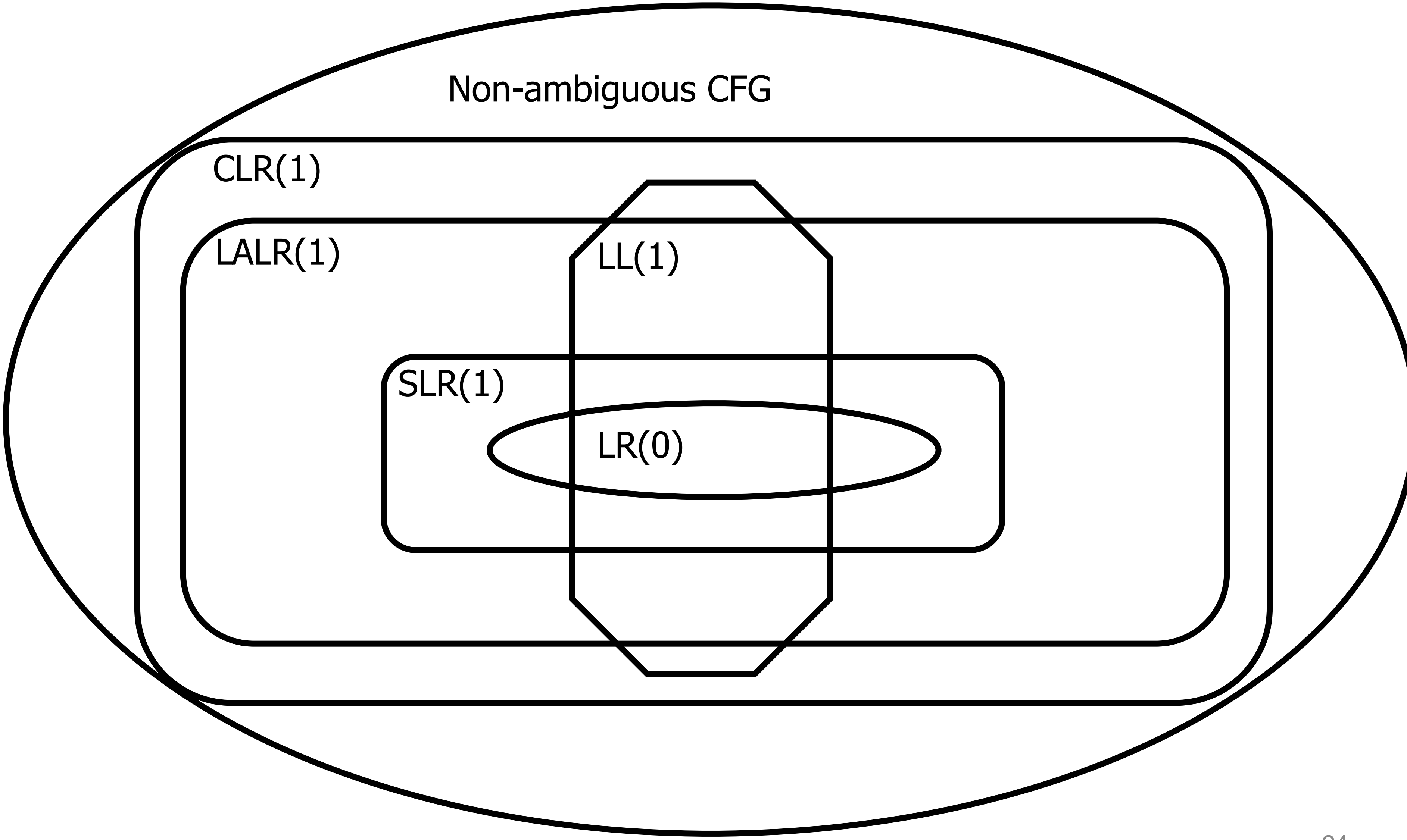
- Top-down (predictive):



- Bottom up (shift-reduce):



Grammar Hierarchy



Top-down Parsing

- Given a grammar G=(V,T,P,S) and a word w
- Goal: derive w using G
- Apply production to leftmost non-terminal
- Pick rule based on next input token
- General grammar
  - more than one option for the next production based on a token
- Restricted grammars LL(1)
  - know exactly which **single rule** to apply
  - may require some **lookahead** to decide

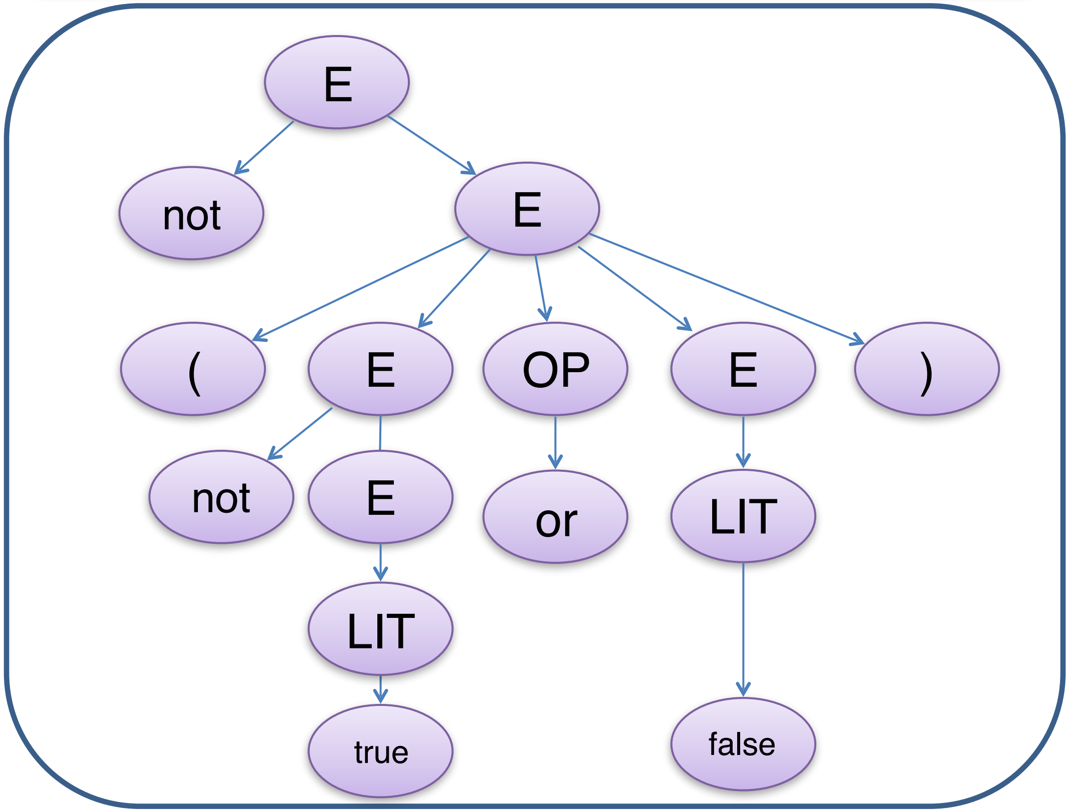
Recursive descent parsing

- Define a function for every symbol
- Terminal: match with next input token
- Non-terminal
  - find applicable production rule
  - if there are several applicable productions, use lookahead to chose one
  - recursively call other functions

Example: boolean expressions

not (not true or false)

```
E → LIT | (E OP E) | not E
LIT → true | false
OP → and | or | xor
```



E =>
not E =>
not (E OP E) =>
not (not E OP E) =>
not (not LIT OP E) =>
not (not true OP E) =>
not (not true or E) =>
not (not true or LIT) =>
not (not true or false)

Production to apply is known from next input token

Recursive descent: terminals

```
void match(token t) {
  if (current == t)
    current = next_token();
  else
    error;
}
```

- Variable `current` holds the current input token

Recursive descent: non-terminals

```
E → LIT | (E OP E) | not E
LIT → true | false
OP → and | or | xor
```

```
void E() {
  if (current ∈{TRUE, FALSE})    // E → LIT
    LIT();
  else if (current == LPAREN)    // E → ( E OP E )
   match(LPARENT); E(); OP(); E(); match(RPAREN);
  else if (current == NOT)       // E → not E
   match(NOT); E();
  else
   error;
 }
void LIT() {
  if (current == TRUE) match(TRUE);          // E → true
  else if (current == FALSE) match(FALSE);   // E → false
  else error;
}
```

Recursive descent: non-terminals

```
E → LIT
  | (E OP E)
  | not E
LIT → true
    | false
OP → and
   | or
   | xor
```

```
void E() {
        if (current ∈ {TRUE, FALSE})                    LIT();
        else if (current == LPAREN)                     match(LPARENT);
                                                        E(); OP(); E();
                                                        match(RPAREN);
        else if (current == NOT)                   match(NOT); E();
        else error;
}
```

```
void LIT() {
        if (current == TRUE)                    match(TRUE);
        else if (current == FALSE)  match(FALSE);
        else                                    error;
}
```

```
void OP() {
        if (current == AND)                     match(AND);
        else if (current == OR)       match(OR);
        else if (current == XOR)      match(XOR);
        else                                    error;
}
```

Semantic actions

- Action to perform on each production rule
- Example: build the parse tree
  - every function returns an object of type Node
  - every Node maintains a list of children
  - function calls can add new children

Building the parse tree

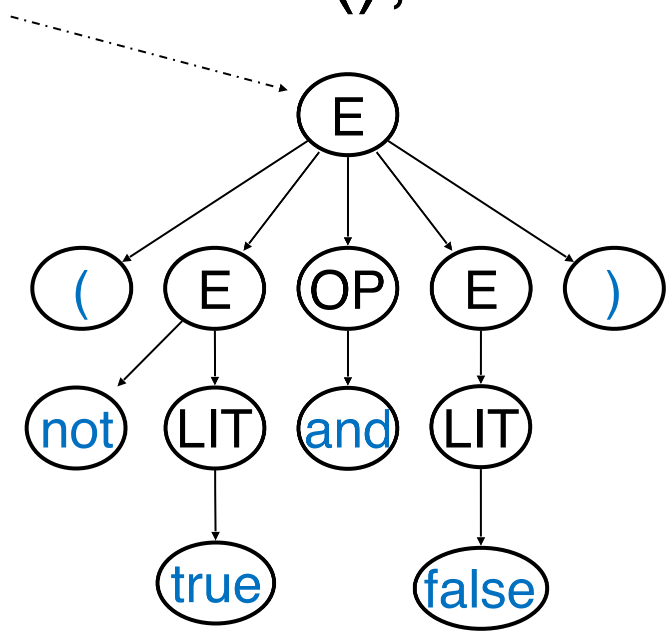
```
Node E() {
  result = new Node();
  result.name = “E”;
  if (current ∈ {TRUE, FALSE}) // E → LIT
    result.addChild(LIT());
  else if (current == LPAREN)  // E → ( E OP E )
    result.addChild(match(LPARENT));
    result.addChild(E());
    result.addChild(OP());
    result.addChild(E());
    result.addChild(match(RPAREN));
  else if (current == NOT)     // E → not E
    result.addChild(match(NOT));
    result.addChild(E());
  else error;
    return result;
}
```

Example

not (not true or false)

```
E → LIT | (E OP E) | not E
LIT → true | false
OP → and | or | xor
```

- Node treeRoot = E();



Recursive Descent

- How do you pick the right A-production?
  - generally, try them all and use **backtracking**
  - special cases: use **lookahead**

```
void A() {
  choose an A-production, A -> X₁X₂…Xₖ;
  for (i=1; i≤k; i++) {
    if (Xi is a nonterminal)
      call procedure Xi();
    else if (Xi == current)
      advance input;
    else
       report error;
  }
}
```

Recursive descent parser: example

```
E → LIT | (E OP E) | not E
LIT → true | false
OP → and | or | xor
```

```
void E() {
  if (current ∈{TRUE, FALSE})    // E → LIT
    LIT();
  else if (current == LPAREN)    // E → ( E OP E )
   match(LPARENT); E(); OP(); E(); match(RPAREN);
  else if (current == NOT)       // E → not E
   match(NOT); E();
  else
   error;
 }
```

How to pick the correct production without backtracking?

Recursive descent: are we done?

```
term → ID | indexed_elem
indexed_elem → ID [ expr ]
```

- What happens for input of the form
  ID [expr]
- The function for indexed_elem will never be tried…

Recursive descent: are we done?

```
S → A a b
A → a | ε
```

- What happens for input “ab” ?
- What happens if you flip order of alternatives and try “aab”?

```
int S() {
   return A() && match(token('a')) && match(token('b'));
}
int A()  {
  return match(token('a')) || 1;
}
```

Recursive descent: are we done?

E → E – term | term

- What happens with this procedure?
- Recursive descent parsers cannot handle **left-recursive** grammars

```
int E() {
    return E() && match(token('-')) && term() || term();
}
```

Figuring out when it works…

```
term → ID | indexed_elem
indexed_elem → ID [ expr ]
```

```
S → A a b
A → a | ε
```

```
E → E – term | term
```

3 examples where we got into trouble with our recursive descent approach

FIRST sets

- Property of a grammar: we can determine the next rule using a single lookahead
- For every production rule A → α
  - FIRST(α) is tokens that α can start with
  - every token that can appear first in a derivation for α
- Boolean expressions example
  - FIRST(LIT) = { true, false }
  - FIRST((E OP E)) = { (}
  - FIRST (not E) = { not }

```
E → LIT | (E OP E) | not E
LIT → true | false
OP → and | or | xor
```

FIRST sets: lookahead

- No intersection between FIRST sets:
we can always pick a single rule
- Predict the alternative $A \rightarrow \alpha$ when the lookahead token is in the set FIRST(α)

- If the FIRST sets intersect, we may need longer lookahead
- LL(k) is a class of grammars in which rule can be determined using a lookahead of k tokens
- LL(1) is an important and useful class

FOLLOW sets

- What do we do with nullable alternatives?
- Example: to select the correct production, we need to know what comes after A
- For every production rule N → α
  FOLLOW(N): tokens that can immediately follow A
- Example
  - FOLLOW(A)={ b }
  - FOLLOW(S)=FOLLOW(B)={ }

```
S → AB | c
A → a  | ε
```

FOLLOW sets: lookahead

- No intersection between FIRST and FOLLOW
- Predict the alternative $N \rightarrow \alpha$ when
  - the lookahead token is in the set FIRST(α)
  - or α is nullable and lookahead is in FOLLOW(N)

LL(k) Grammars

- A grammar is in the class LL(K) when it can be derived via:
  - top down derivation
  - scanning the input from left to right (L)
  - producing the leftmost derivation (L)
  - with lookahead of k tokens (k)
- A language is said to be LL(k) when it has an LL(k) grammar

Back to our 1st example

```
term → ID | indexed_elem
indexed_elem → ID [ expr ]
```

- FIRST(term) = { ID }
- FIRST(indexed_elem) = { ID }

- FIRST/FIRST conflict
- This grammar is not in LL(1)

Can we “fix” it?

Left factoring

- Rewrite the grammar to be in LL(1)

```
term → ID | indexed_elem
indexed_elem → ID [ expr ]
```

```
term → ID after_ID
after_ID → [ expr ] | ε
```

Intuition: just like factoring x*y + x*z into x*(y+z)

Left factoring: another example

```
S → if E then S else S
  | if E then S
  | T
```

```
S → if E then S S'
  | T
S' → else S | ε
```

Back to our 2nd example

```
S → A a b
A → a | ε
```

- Select a rule for A with a in the look-ahead
- Should we pick $A \rightarrow a$ or $A \rightarrow e$?

- FIRST(S) = { a } FOLLOW(S) = { }
- FIRST(A) = { a } FOLLOW(A) = { a }

- FIRST/FOLLOW conflict
- The grammar is not in LL(1)

Can we “fix” it?

An equivalent grammar via Substitution

```
S → A a b
A → a | ε
```

Substitute A in S

```
S → a a b | a b
```

Left factoring

```
S → a after_A
after_A → a b | b
```

So Far

- Can determine if a grammar is in LL(1) using FIRST and FOLLOW sets
- Have some techniques for modifying a grammar to find an equivalent grammar in LL(1)
  - left factoring
  - substitution
- Now let's look at the 3rd example and present one more such technique

Back to our 3rd example

```
E → E – term | term
```

- Left recursion cannot be handled with a bounded lookahead
- What can we do?
- Any grammar with left recursion has an equivalent grammar without left recursion

Left recursion removal

$N \rightarrow N\alpha \mid \beta$

G1

$N \rightarrow \beta N'$
$N' \rightarrow \alpha N' \mid \varepsilon$

G2

- L(G1) = β, βα, βαα, βααα, …
- L(G2) = same

- For our 3rd example:

```
E → E – term | term
```

```
E  → term TE
TE → - term TE | ε
```

LL(k) Parsers

- Recursive Descent
  - manual construction
  - uses recursion

- Wanted
  - parser that can be generated automatically
  - does not use recursion

LL(k) parsing with pushdown automata

- Prediction stack
- Input stream
- Transition table
  - from (nonterminal and token)
    to production alternative
  - entry indexed by nonterminal N and token t
    contains the alternative of N **predicated** when
    current input starts with t

LL(k) parsing with pushdown automata

- Two possible moves for next input token t
- **Prediction**: when top of stack is nonterminal N
  - if table[N,t] is not empty, pop N and push table[N,t] on prediction stack
  - otherwise, syntax error
- **Match**: when top of prediction stack is a terminal T
  - if (t == T) then pop T and consume t
  - otherwise, syntax error
- **Termination**
  - if input is empty and prediction stack is empty, then success
  - otherwise, syntax error

Example transition table

(1) E → LIT
(2) E → (E OP E)
(3) E → not E
(4) LIT → true
(5) LIT → false
(6) OP → and
(7) OP → or
(8) OP → xor

Which rule should be used

Input tokens

Nonterminals

	(	)	not	true	false	and	or	xor	$
E	2		3	1	1				
LIT				4	5				
OP						6	7	8	

Simple example

aacbb$

`A → aAb | c`

Input suffix	Stack content (top on left)	Move
aacbb$	A$	predict(A,a) = A → aAb
aacbb$	aAb$	match(a,a)
acbb$	Ab$	predict(A,a) = A → aAb
acbb$	aAbb$	match(a,a)
cbb$	Abb$	predict(A,c) = A → c
cbb$	cbb$	match(c,c)
bb$	bb$	match(b,b)
b$	b$	match(b,b)
$	$	match($,$) – success

	a	b	c
A	A → aAb		A → c

Transition table construction

- Builds on FIRST and FOLLOW

Example: bad word

abcbb$

$A \rightarrow aAb \mid c$

Input suffix	Stack content	Move
abcbb$	A$	predict(A,a) = A → aAb
abcbb$	aAb$	match(a,a)
bcbb$	Ab$	predict(A,b) = ERROR

	a	b	c
A	A → aAb		A → c

Error handling

- Types of errors
  - lexical errors
  - syntax errors
  - semantic errors (e.g., type mismatch)
  - logical errors (e.g., infinite loop)

- Requirements
  - Report the error clearly
  - Recover and continue so more errors can be discovered
  - Efficiency

Error handling and recovery

```
x = a * (p+q * ( -b * (r-s);
```

- Where should we report the error?
- The valid prefix property
- Recovery is tricky
- Heuristics
  - dropping tokens
  - skipping to semicolon,
  - ...

Error handling in LL parsers

c$

`S → a c | b S`

Input suffix	Stack content	Move
c$	S$	predict(S,c) = ERROR

- Now what?
- Predict bS anyway “missing token b inserted in line XXX”

	a	b	c
S	S → a c	S → bS	

Error handling in LL parsers

c$

S → a c | b S

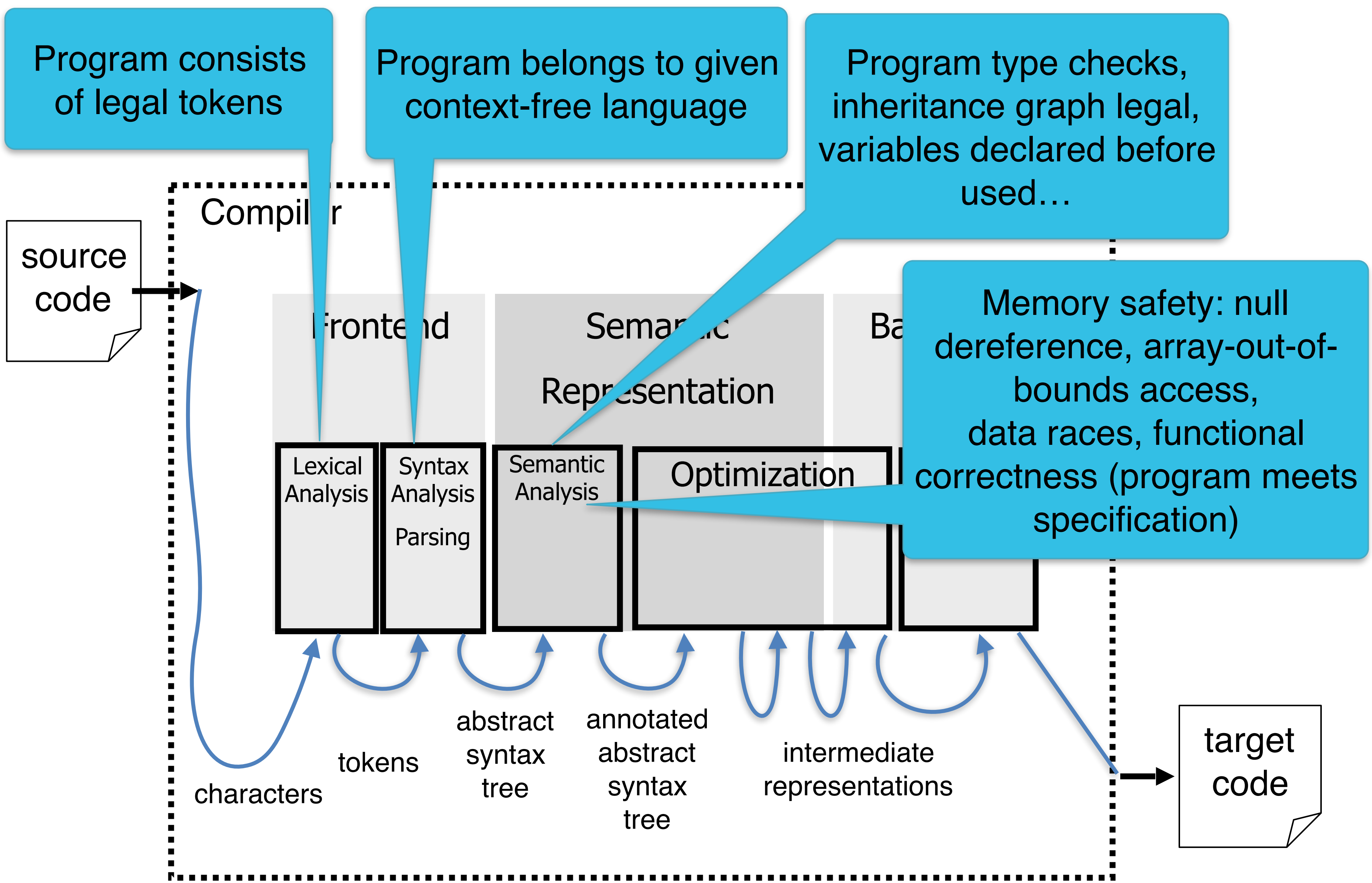
Input suffix	Stack content	Move
bc$	S$	predict(S,b) = S → bS
bc$	bS$	match(b,b)
c$	S$	Looks familiar?

- Result: infinite loop
- Requires more systematic treatment

	a	b	c
S	S → a c	S → bS	

Error handling examples

- Panic mode (or acceptable-set method): drop tokens until reaching a **synchronizing token**
  - semicolon, right parenthesis, end of file, ...
- Phrase-level recovery: attempting **local changes**
  - replace “,” with “;”
  - eliminate or add a “;”
- Error production: anticipate errors and automatically handle them by **adding them to the grammar**
- Global correction: find the minimum modification to the program that will make it derivable in the grammar
  - Not a practical solution…



Program consists of legal tokens
Program belongs to given context-free language
Program type checks, inheritance graph legal, variables declared before used…
Memory safety: null dereference, array-out-of-bounds access, data races, functional correctness (program meets specification)
source code
Compil r
Fronte d
Sema ic
Repr sentation
Ba
Lexical Analysis
Syntax Analysis
Parsing
Semantic Analysis
Optimization
characters
tokens
abstract syntax tree
annotated abstract syntax tree
intermediate representations
target code

What formalism should we use?

- Expressivity
- Computational complexity

finite languages	regular languages	context-free languages	context-sensitive languages	…
finite state automata		pushdown automata	?	

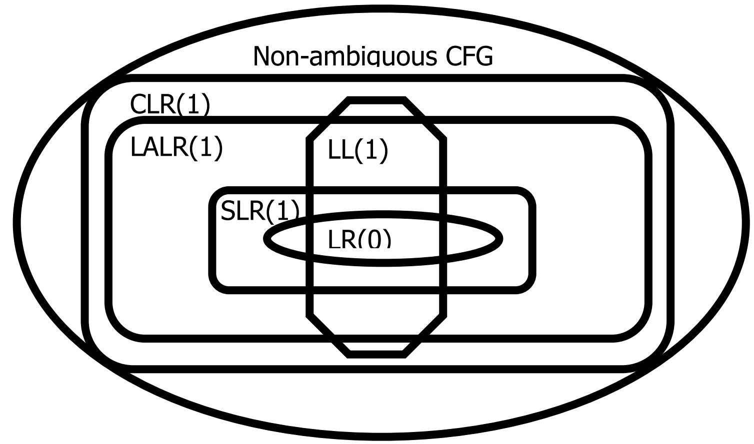
Do we really need lexical analysis?

- Regular languages is a subset of context-free languages
- Why separate lexical analysis from syntactic analysis?

- Not strictly necessary, but….
- Leads to more efficient analysis
- Simplifies parser definition
- Simplifies language definition
- Enhances compiler portability
  - for example, input alphabet concerns restricted to lexer

Theoretical limitations on grammars

- Important goals
  - unambiguous language
  - efficient parser
- Restrict the grammar
- Some languages do not fit in
  - conflicts during parser generation
  - inputs do not parse as intended



Traditional parser generators

- Rewrite grammar to appease the generator
- Alter the language to fit parser generator
- Modify the generated code
  - language features
  - performance
  - error handling
- User needs to understand a lot about how the generator works

Traditional parser generators

- Intention: decouple grammar description from implementation language
- Reality: grammar rules mixed with code fragments
    - pain to develop and maintain
    - can’t reuse grammar for multiple projects
    - can’t port to other implementation languages
    - parse tree structure is hard to modify
- Antlr: visitors and listeners

Parsing in the real world

- Hand-written recursive descent parsers in production compilers
  gcc, llvm, camlopt, …

Summary

- Parsing
  - Top-down or bottom-up
- Top-down parsing
  - Recursive descent
  - LL(k) grammars
  - LL(k) parsing with pushdown automata
- LL(K) parsers
  - Cannot deal with left recursion
  - Left-recursion removal might complicated grammar