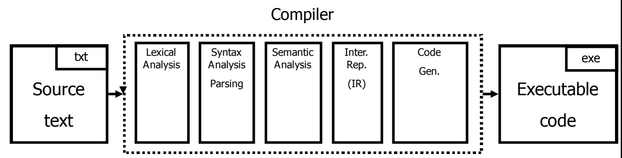
Lecture 03 – Syntax analysis: top-down parsing

THEORY OF COMPILATION

Eran Yahav

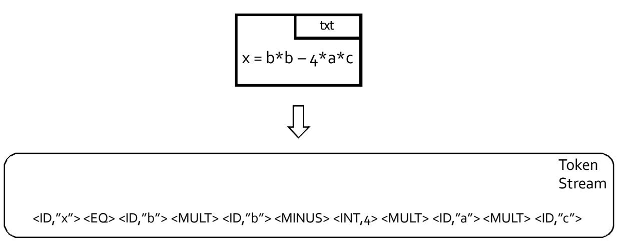
1

You are here



2

Last Week: from characters to tokens



3

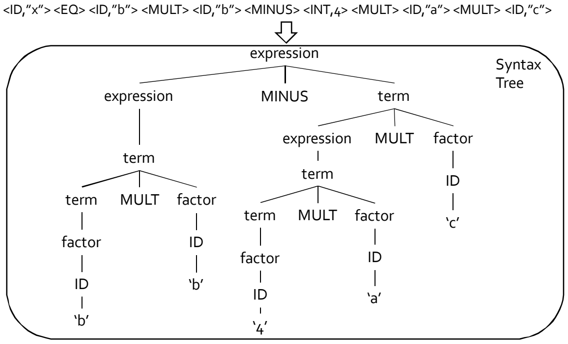
Last Week: Regular Expressions

Basic Patterns	**Matching**
x	The character x
.	Any character, usually except a new line
[xyz]	Any of the characters x,y,z
Repetition Operators	
R?	An R or nothing (=optionally an R)
R*	Zero or more occurrences of R
R+	One or more occurrences of R
Composition Operators	
R1R2	An R1 followed by R2
R1|R2	Either an R1 or R2
Grouping	
(R)	R itself

4

Today: from tokens to AST

<ID,"x"> <EQ> <ID,"b"> <MULT> <ID,"b"> <MINUS> <INT,4> <MULT> <ID,"a"> <MULT> <ID,"c">



Lexical Analysis | Syntax Analysis | Sem. Analysis | Inter. Rep. | Code Gen.

Parsing

- Goals
  - Is a sequence of tokens a valid 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
- T – terminals (tokens)
- P – derivation rules
  - Each rule of the form $V \rightarrow (T \cup V)^*$
- S – initial symbol

Why do we need context free grammars?

$S \rightarrow SS$

$S \rightarrow (S)$

$S \rightarrow ()$

Example

S → S;S

S → id := E

E → id | E **+** E | E ***** E | **(** E **)**

V = { S, E }
T = { id, '+', '*', '(', ')'}

Derivation

input

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

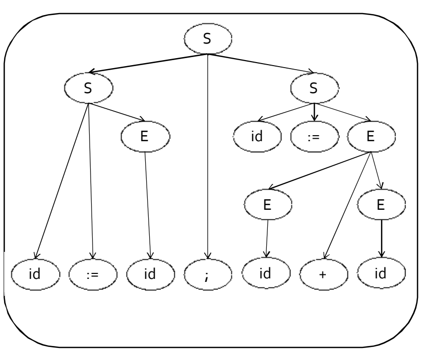
grammar

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

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


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
```



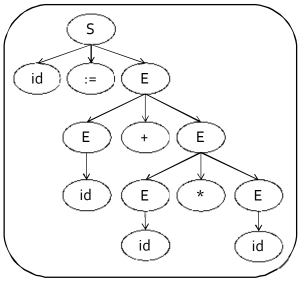
Questions

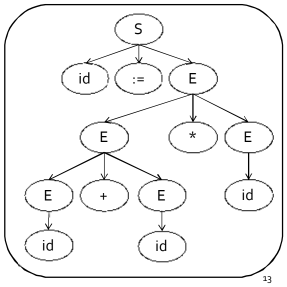
- How did we know which rule to apply on every step?
- Does it matter?
- Would we always get the same result?

Ambiguity

x := y+z*w

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





13

Leftmost/rightmost Derivation

- Leftmost derivation
  - always expand leftmost non-terminal
- Rightmost derivation
  - Always expand rightmost non-terminal
- Allows us to describe derivation by listing the sequence of rules
  - always know what a rule is applied to
- Orders of derivation applied in our parsers (coming soon)

14

Leftmost Derivation

x := z;
y := x + z

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

```
              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
```

15

Rightmost Derivation

x := z;
y := x + z

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

```
              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
```

16

Bottom-up Example

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
 id := E; id := id + id
                            S → id := E
     S ; id := id + id
                            E → id
     S ; id := E + id
                            E → id
     S ; id := E + E
                            E → E + E
     S ; id := E
                            S → id := E
     S ;  S
                            S → S;S
       S
```

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

Parsing

- A context free language can be recognized by a non-deterministic pushdown automaton
- Parsing can be seen as a search problem
  - Can you find a derivation from the start symbol to the input word?
  - Easy (but very expensive) to solve with backtracking
- 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
  - We will sacrifice generality for efficiency

"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
```

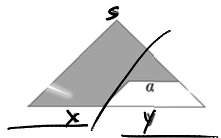
(not a parse tree… a search for the parse tree by exhaustively applying all rules)

Efficient Parsers

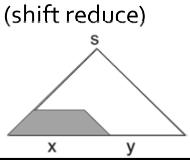
- Top-down (predictive)
  - Construct the leftmost derivation
  - Apply rules "from left to right"
  - Predict what rule to apply based on nonterminal and token
- Bottom up (shift reduce)
  - Construct the rightmost derivation
  - Apply rules "from right to left"
  - Reduce a right-hand side of a production to its non-terminal

Efficient Parsers

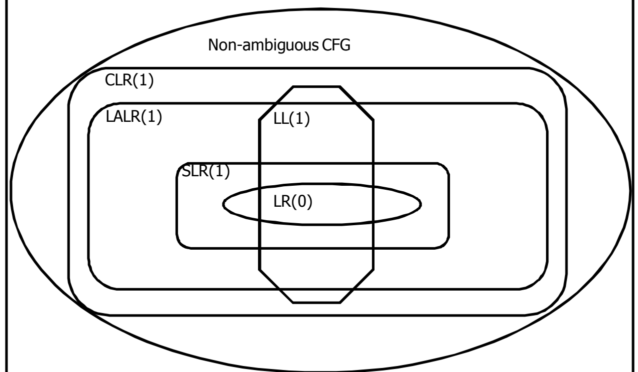
- Top-down (predictive parsing)



- Bottom-up (shift reduce)



Grammar Hierarchy



Top-down Parsing

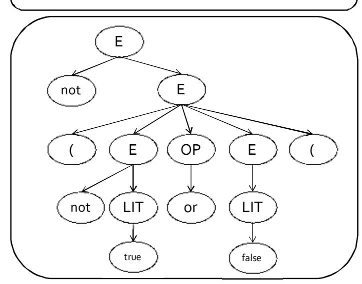
- Given a grammar G=(V,T,P,S) and a word w
- Goal: derive w using G
- Idea
  - Apply production to leftmost nonterminal
  - Pick production rule based on next input token
- General grammar
  - More than one option for choosing the next production based on a token
- Restricted grammars (LL)
  - Know exactly which single rule to apply
  - May require some lookahead to decide

Boolean Expressions Example

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 Parsing

- Define a function for every nonterminal
- Every function work as follows
  - Find applicable production rule
  - Terminal function checks match with next input token
  - Nonterminal function calls (recursively) other functions
- If there are several applicable productions for a nonterminal, use lookahead

Matching tokens

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

- Variable current holds the current input token

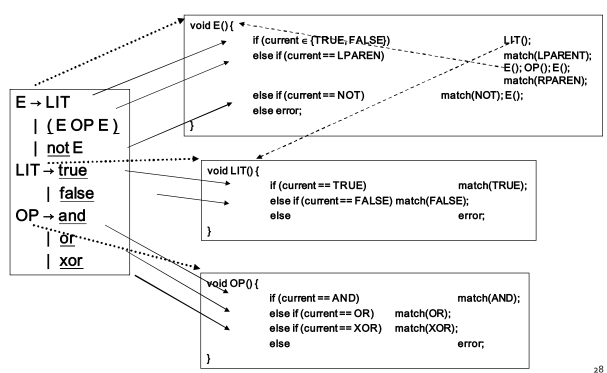
functions for nonterminals

```
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);
   else if (current == FALSE) match(FALSE);
   else error;
 }
```


functions for nonterminals



Adding semantic actions

- Can add an action to perform on each production rule
- Can 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;
}
```


Recursive Descent

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

- How do you pick the right A-production?
- Generally – try them all and use backtracking
- In our case – use lookahead

Recursive descent: are we done?

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

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

Recursive descent: are we done?

```
S → A a b
A → a | ε
```

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

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

Recursive descent: are we done?

```
E → E - term
```

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

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

Figuring out when it works…

1

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

2

```
S → A a b
A → a | ε
```

3

```
E → E - term
```

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

FIRST sets

- For every production rule A→α
  - FIRST(α) = all terminals that α can start with
  - i.e., every token that can appear as first in α under some derivation for α
- In our Boolean expressions example
  - FIRST(LIT) = { true, false }
  - FIRST((E OP E)) = { '(' }
  - FIRST (not E) = { not }
- No intersection between FIRST sets => can always pick a single rule
- If the FIRST sets intersect, may need longer lookahead
  - LL(k) = class of grammars in which production 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?
  - Use what comes afterwards to predict the right production
- For every production rule A⟶α
  - FOLLOW(A) = set of tokens that can immediately follow A
- Can predict the alternative A_k for a non-terminal N when the lookahead token is in the set
  - FIRST(A_k) ∪ (if A_k is nullable then FOLLOW(N))

37

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

38

Back to our 1st example

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

- FIRST(ID) = { ID }
- FIRST(indexed_elem) = { ID }
- FIRST/FIRST conflict

39

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)

40

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
```

- FIRST(S) = { 'a' }, FOLLOW(S) = { }
- FIRST(A) = { 'a' ε }, FOLLOW(A) = { 'a' }

- FIRST/FOLLOW conflict

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
```


Back to our 3rd example

```
E → E - term
```

- Left recursion cannot be handled with a bounded lookahead

- What can we do?

Left recursion removal

N → Nα | β ⇨ N → βN′ ; N′ → αN′ | ε

G1 G2

- L(G1) = β, βα, βαα, βααα, …
- L(G2) = same
- For our 3rd example:

```
E → E - term
```

⇨

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


LL(k) Parsers

- Recursive Descent
  - Manual construction
  - Uses recursion
- Wanted
  - A parser that can be generated automatically
  - Does not use recursion

LL(k) parsing with pushdown automata

- Pushdown automaton uses
  - Prediction stack
  - Input stream
  - Transition table
    - nonterminals x tokens -> production alternative
    - Entry indexed by nonterminal N and token t contains the alternative of N that must be predicated when current input starts with t

LL(k) parsing with pushdown automata

- Two possible moves
  - Prediction
    - When top of stack is nonterminal N, pop N, lookup table[N,t]. If table[N,t] is not empty, push table[N,t] on prediction stack, otherwise – syntax error
  - Match
    - When top of prediction stack is a terminal T, must be equal to next input token t. If (t == T), pop T and consume t. If (t ≠ T) syntax error
- Parsing terminates when prediction stack is empty. If input is empty at that point, 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	

49

Simple Example

aacbb$

`A → aAb | c`

Input suffix	Stack content	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

50

Simple Example

abcbb$

`A → aAb | 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

51

Error Handling

- Mentioned last time
  - Lexical errors
  - Syntax errors
  - Semantic errors (e.g., type mismatch)

52

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 for dropping tokens, skipping to semicolon, etc.

53

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	

54

Error Handling in LL Parsers

c$

S → a c | b S

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

- Result: infinite loop

	a	b	c
S	S → a c	S → bS	

55

Error Handling

- Requires more systematic treatment
- Enrichment
  - Acceptable-set method
  - Not part of course material

56

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 result with complicated grammar

57

Coming up next time

- More syntax analysis

58