Compilation

0368-3133 (Semester A, 2013/14)

Lecture 3: Syntax Analysis
(Top-Down Parsing)

Modern Compiler Design: Chapter 2.2

Noam Rinetzky

Slides credit: Roman Manevich, Mooly Sagiv, Jeff Ullman, Eran Yahav

Admin

- Slides: *http://www.cs.tau.ac.il/~maon/…*
  - All info: Moodle
- Next week: Dan David 001
  - Vote …
- Mobiles …

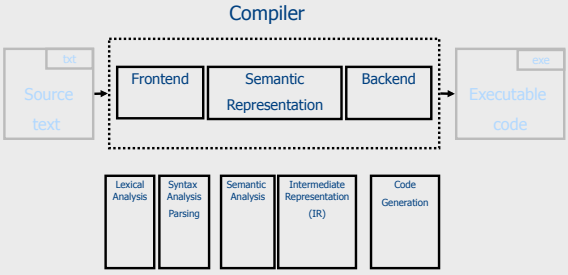
What is a Compiler?

"A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

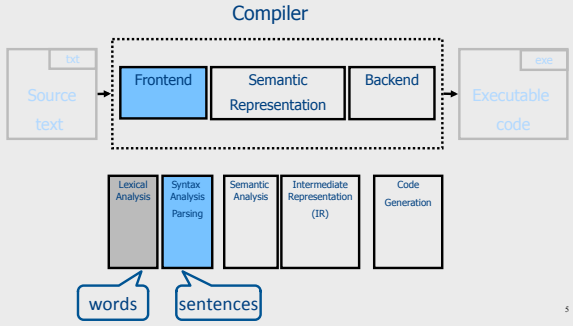
The most common reason for wanting to transform source code is to create an executable program."

--Wikipedia

Conceptual Structure of a Compiler



Conceptual Structure of a Compiler



5

Lexical Analysis Refresher

- Example: Lang. of fully parenthesized exprs

Expr → Num | LP Expr Op Expr RP
Num → Dig | Dig Num
Dig → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
LP → '('
RP → ')'
Op → '+' | '*'

((23 + 7) * 19)

6

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language: Expr → Num | LP Expr Op Expr RP

Regular languages:
Num → Dig | Dig Num
Dig → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
LP → '('
RP → ')'
Op → '+' | '*'

((23 + 7) * 19)

7

What does Lexical Analysis do?

- Language: fully parenthesized expressions

Context free language: Expr → Num | LP Expr Op Expr RP

Regular languages:
Num → Dig | Dig Num
Dig → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
LP → '('
RP → ')'
Op → '+' | '*'

	Token	Token	...						
Value	(	(	23	+	7	)	*	19	)
Kind	LP	LP	Num	Op	Num	RP	Op	Num	RP

8

Scanners: Spec. & Impl.

```
if                                  { return IF; }
[a-z][a-z0-9]*                      { return ID; }
[0-9]+                              { return NUM; }
[0-9]"."[0-9]*|[0-9]*"."[0-9]+      { return REAL; }
(\-\-[a-z]*\n)|(" "|\n|\t)          { ; }
.                                   { error(); }
```

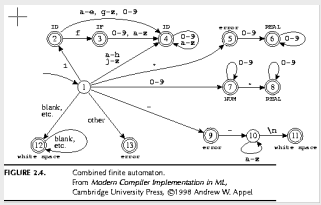


FIGURE 2.4. Combined finite automaton.
From *Modern Compiler Implementation in ML*,
Cambridge University Press, ©1998 Andrew W. Appel

Scanner usage of DFSM goes beyond language recognition

9

Frontend: Scanning & Parsing

program text ((23 + 7) * x)

Lexical Analyzer

token stream

(	(	23	+	7	)	*	x	)
LP	LP	Num	OP	Num	RP	OP	Id	RP

Grammar:
E → … | Id
Id → 'a' | … | 'z'

Parser

syntax error

valid

Op(*)

Op(+) Id(b)

Num(23) Num(7)

Abstract Syntax Tree

10

From scanning to parsing

program text ((23 + 7) * x)

Lexical Analyzer

token stream

(	(	23	+	7	)	*	x	)
LP	LP	Num	OP	Num	RP	OP	Id	RP

Grammar:
E → … | Id
Id → 'a' | … | 'z'

Parser

syntax error

valid

Op(*)

Op(+) Id(b)

Num(23) Num(7)

Abstract Syntax Tree

11

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?
  - How do you construct the structured representation?
  - Where do you report an error?

12

Lecture Outline

- ✓Role & place of syntax analysis
- Context free languages
  - Grammars
  - Push down automata (PDA) ★
- Predictive parsing
- Error handling
- Semantics actions
- Earley parser

13

Context free languages

- Example: $\{ 0^n1^n \mid n > 0 \}$
- Context Free Grammars (CFG)

  ~
- Push Down Automata (PDA)
  - Non Deterministic Pushdown Automata

14

Context free grammars (CFG)

G = (V,T,P,S)

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

15

What can CFGs do?

- Recognize CFLs
- $S \rightarrow 0T1$
- $T \rightarrow 0T1 \mid \varepsilon$

16

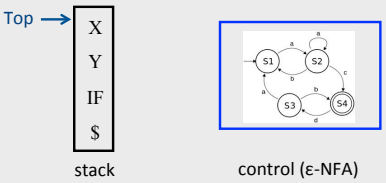
Pushdown Automata

- Nondeterministic PDA is an automaton that defines a CFL language
  - Nondeterministic PDAs define all CFLs
    - Equivalent to the CFG in language-defining power
- The deterministic version models parsers.
  - Most programming languages have deterministic PDAs
  - Efficient implementation

17

Intuition: PDA

- An ε-NFA with the additional power to manipulate **a** stack



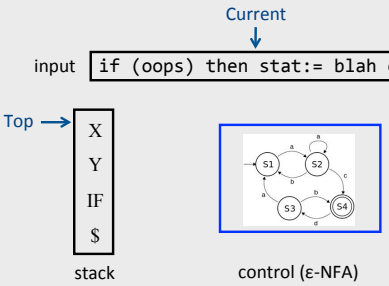
18

Intuition: PDA

- Think of an ε-NFA with the additional power that it can manipulate a stack
- Its moves are determined by:
  - The current state (of its "ε-NFA")
  - The current input symbol (or ε)
  - The current symbol on top of its stack

19

Intuition: PDA



20

Intuition: PDA – (2)

- Moves:
  - Change state
  - Replace the top symbol by 0…n symbols
    - 0 symbols = "pop" ("reduce")
    - 0 < symbols = sequence of "pushes" ("shift")
- Nondeterministic choice of next move

PDA Formalism

- PDA = (A, Σ, Γ, q_0, $, F):
  - A finite set of states (Q, typically).
  - An input alphabet (Σ, typically).
  - A stack alphabet (Γ, typically).
  - A transition function (δ, typically).
  - A start state (q_0, in Q, typically).
  - A start symbol ($, in Γ, typically).
  - A set of final states (F ⊆ Q, typically).

The Transition Function

- $\delta(q, a, X) = \{ (p_1, \sigma_1), \ldots, (p_n, \sigma_n)\}$
  - Input: triplet
    - A state $q \in Q$
    - An input symbol $a \in \Sigma$ or ε
    - A stack symbol $X \in \Gamma$
  - Output: set of 0 … k actions of the form (p, σ)
    - A state $q \in Q$
    - σ a sequence $X_1 \cdots X_n \in \Gamma^*$ of stack symbols

Actions of the PDA

- Say $(p, \sigma) \in \delta(q, a, X)$
  - If the PDA is in state q and X is the top symbol and a is at the front of the input
  - Then it can
    - Change the state to p.
    - Remove a from the front of the input
      - (but a may be ε).
    - Replace X on the top of the stack by σ.

Example: Deterministic PDA

- Design a PDA to accept $\{0^n1^n \mid n > 1\}$.
- The states:
  - q = start state. We are in state q if we have seen only 0's so far.
  - p = we've seen at least one 1 and may now proceed only if the inputs are 1's.
  - f = final state; accept.

25

Example: Stack Symbols

- $ = start symbol. Also marks the bottom of the stack, so we know when we have counted the same number of 1's as 0's.
- X = "counter", used to count the number of 0's seen on the input.

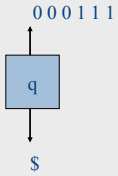
26

Example: Transitions

- $\delta(q, 0, \$) = \{(q, X\$)\}$.
- $\delta(q, 0, X) = \{(q, XX)\}$.
  - These two rules cause one X to be pushed onto the stack for each 0 read from the input.
- $\delta(q, 1, X) = \{(p, \varepsilon)\}$.
  - When we see a 1, go to state p and pop one X.
- $\delta(p, 1, X) = \{(p, \varepsilon)\}$.
  - Pop one X per 1.
- $\delta(p, \varepsilon, \$) = \{(f, \$)\}$.
  - Accept at bottom.

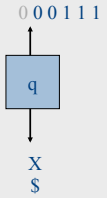
27

Actions of the Example PDA



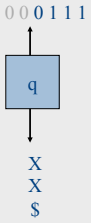
28

Actions of the Example PDA



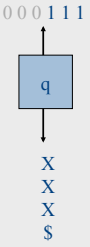
29

Actions of the Example PDA



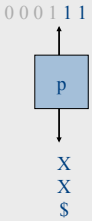
30

Actions of the Example PDA



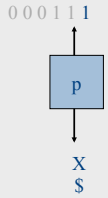
31

Actions of the Example PDA



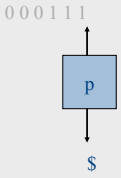
32

Actions of the Example PDA



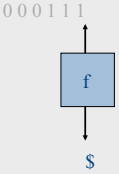
33

Actions of the Example PDA



34

Actions of the Example PDA



35

Example: Non Deterministic PDA

- A PDA that accepts palindromes
  - L {pp' ∈ Σ* | p'=reverse(p)}

36

Instantaneous Descriptions

- We can formalize the pictures just seen with an instantaneous description (ID).
- A ID is a triple (q, w, α), where:
  - q is the current state.
  - w is the remaining input.
  - α is the stack contents, top at the left.
- Define a transition relation between IDs

37

Context free grammars

G = (V,T,P,S)

- V – non terminals (syntactic variables)
- T – terminals (tokens)
- P – derivation rules
  - Each rule of the form V → (T ∪ V)*
- S – start symbol

38

Example grammar

S → S **;** S
S → id **:=** E
E → **id**
E → **num**
E → E **+** E

S shorthand for Statement

E shorthand for Expression

39

CFG terminology

S → S **;** S
S → id **:=** E
E → **id**
E → **num**
E → E **+** E

Symbols:
Terminals (tokens): **; := ( ) id num print**
Non-terminals: S E L

Start non-terminal: S
Convention: the non-terminal appearing in the first derivation rule

Grammar productions (rules)
N → μ

40

CFG 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

41

Derivations

- Show that a sentence ω is in a grammar G
  - Start with the start symbol
  - Repeatedly replace one of the non-terminals by a right-hand side of a production
  - Stop when the sentence contains only terminals
- Given a sentence αNβ and rule N→μ
  αNβ => αμβ
- ω is in L(G) if S =>* ω

42

Derivation

sentence

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

grammar

```
S → S;S
S → id := E | …
E → id | E + E | E * E |…
```

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

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

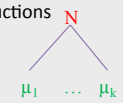
<id,"x"><ASS><id,"z"><SEMI><id,"y"><ASS><id,"x"><PLUS><id,"z">

43

Parse trees: Traces of Derivations

- Tree nodes are symbols, children ordered left-to-right
- Each internal node is non-terminal and its children correspond to one of its productions

N → $\mu_1 \ldots \mu_k$

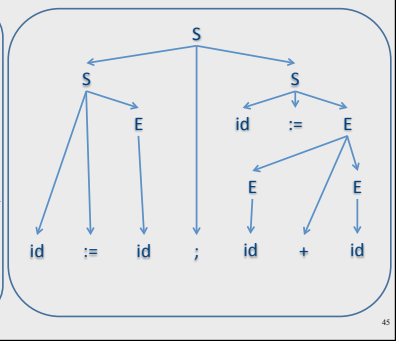


- Root is start non-terminal
- Leaves are tokens
- *Yield* of parse tree: left-to-right walk over leaves

44

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
```



45

Language of a CFG

- A sentence ω is in L(G) (valid program) if
  - There exists a corresponding derivation
  - There exists a corresponding parse tree

46

Questions

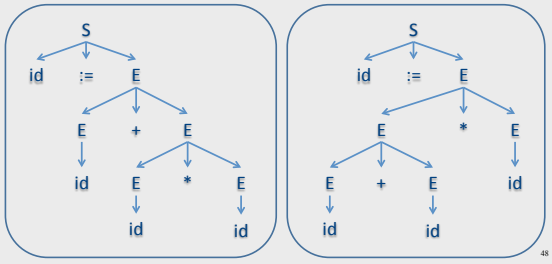
- How did we know which rule to apply on every step?
- Would we always get the same result?
- Does it matter?

47

Ambiguity

x := y+z*w

```
S → S ; S
S → id := E | …
E → id | E + E | E * E | …
```



48

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

49

Leftmost Derivation

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

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

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

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

50

Rightmost Derivation

```
<id,"x"> ASS <id,"z"> ;
<id,"y"> ASS <id,"x"> PLUS
<id,"z">
```

```
S → S;S
S → id := E | …
E → id | E + E | E * E | …
```

```
S
S ; S
S ; id := E
S ; id := E + E
S ; id := E + id
S ; id := id + id
id := E ; id := id + id
id := id ; id := id + id
<id,"x"> ASS <id,"z"> ; <id,"y"> ASS <id,"x"> PLUS <id,"z">
```

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

51

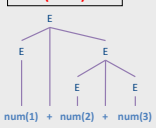
Sometimes there are two parse trees

Arithmetic expressions:

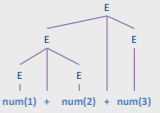
E → id
E → num
E → E + E
E → E * E
E → (E)

1 + 2 + 3

"1 + (2 + 3)"



"(1 + 2) + 3"



Leftmost derivation

E
E + E
num + E
num + E + E
num + num + E
num + num + num

Rightmost derivation

E
E + E
E + num
E + E + num
E + num + num
num + num + num

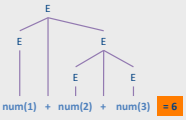
52

Is ambiguity a problem?

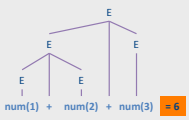
Arithmetic expressions:
E → id
E → num
E → E + E
E → E * E
E → (E)

1 + 2 + 3

"1 + (2 + 3)"



"(1 + 2) + 3"



Leftmost derivation
E
E + E
num + E
num + E + E
num + num + E
num + num + num

Rightmost derivation
E
E + E
E + num
E + E + num
E + num + num
num + num + num

53

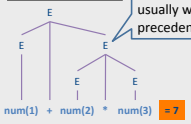
Problematic ambiguity example

Arithmetic expressions:
E → id
E → num
E → E + E
E → E * E
E → (E)

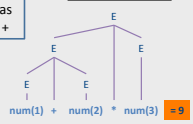
1 + 2 * 3

"1 + (2 * 3)"

This is what we usually want: * has precedence over +



"(1 + 2) * 3"



Leftmost derivation
E
E + E
num + E
num + E * E
num + num * E
num + num * num

Rightmost derivation
E
E * E
E * num
E + E * num
E + num * num
num + num * num

54

Ambiguous grammars

- A grammar is *ambiguous* if there exists a sentence for which there are
  - Two different leftmost derivations
  - Two different rightmost derivations
  - Two different parse trees
- Property of *grammars*, not *languages*

- Some languages are inherently ambiguous – no unambiguous grammars exist
- No algorithm to detect whether arbitrary grammar is ambiguous

55

Drawbacks of ambiguous grammars

- Ambiguous semantics
  - 1 + 2 * 3 = 7 or 9
- Parsing complexity
- May affect other phases
- Solutions
  - Transform grammar into non-ambiguous
  - Handle as part of parsing method
    - Using special form of "precedence"

56

Transforming ambiguous grammars to non-ambiguous by layering

Ambiguous grammar
E → E + E
E → E * E
E → id
E → num
E → (E)

Unambiguous grammar
E → E + T — Layer 1
E → T
T → T * F — Layer 2
T → F
F → id — Layer 3
F → num
F → (E)

Each layer takes care of one way of composing sub-strings to form a string:
1: by +
2: by *
3: atoms

Let's derive 1 + 2 * 3

57

Transformed grammar: * precedes +

Ambiguous grammar
E → E + E
E → E * E
E → id
E → num
E → (E)

Unambiguous grammar
E → E + T
E → T
T → T * F
T → F
F → id
F → num
F → (E)

Derivation
E
=> E + T
=> T + T
=> F + T
=> 1 + T
=> 1 + T * F
=> 1 + F * F
=> 1 + 2 * F
=> 1 + 2 * 3

Parse tree

58

Transformed grammar: + precedes *

Ambiguous grammar
E → E + E
E → E * E
E → id
E → num
E → (E)

Unambiguous grammar
E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

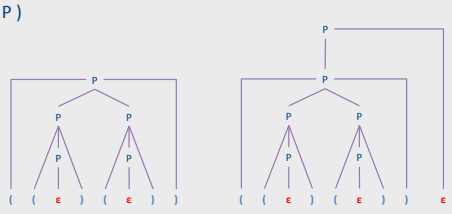
Derivation
E
=> E * T
=> T * T
=> T + F * T
=> F + F * T
=> 1 + F * T
=> 1 + 2 * T
=> 1 + 2 * F
=> 1 + 2 * 3

Parse tree

59

Another example for layering

Ambiguous grammar
P → ε
| P P
| (P)



60

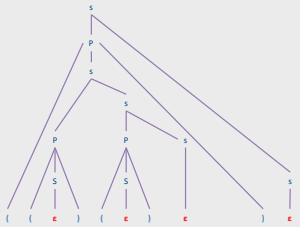
Another example for layering

Ambiguous grammar
P → ε
| P P
| (P)

Unambiguous grammar
S → P S
| ε
P → (S)

Takes care of "concatenation"

Takes care of nesting



61

"dangling-else" example

p. 174

Ambiguous grammar
S → if E then S
S | if E then S else S
| other

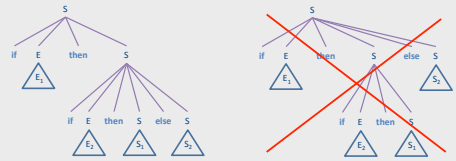
This is what we usually want: match else to closest unmatched then

Unambiguous grammar
?

if E_1 then if E_2 then S_1 else S_2

if E_1 then (if E_2 then S_1 else S_2)

if E_1 then (if E_2 then S_1) else S_2



62

Broad kinds of parsers

- Parsers for arbitrary grammars
  - Earley's method, CYK method O(n^3)
  - Usually, not used in practice (though might change)
- Top-Down parsers
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
  - Predictive: for every non-terminal and k-tokens predict the next production LL(k)
  - Preorder tree traversal
- Bottom-Up parsers
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in a reverse order
  - For every potential right hand side and k-tokens decide when a production is found LR(k)
  - Postorder tree traversal

63

Broad kinds of parsers

- Parsers for arbitrary grammars
  - Earley's method, CYK method O(n^3)
  - Usually, not used in practice (though might change)
- Top-Down parsers
  - Construct parse tree in a top-down matter
  - Find the leftmost derivation
- Bottom-Up parsers
  - Construct parse tree in a bottom-up manner
  - Find the rightmost derivation in a reverse order

64

Intuition: Top-down vs. bottom-up

- Top-down parsing
  - Begin with start symbol
  - Guess the productions
  - Check if parse tree yields user's program
- Bottom-up parsing
  - Begin with the user's program
  - Guess parse subtrees
  - Check if root is the start symbol

65

Intuition: Top-Down Parsing

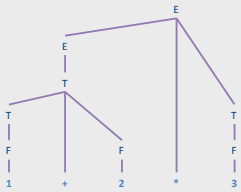
- Begin with start symbol
- Guess the productions
- Check if parse tree yields user's program

66

Intuition: Top-Down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)



67

Intuition: Top-Down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

We need this rule to get the *

E

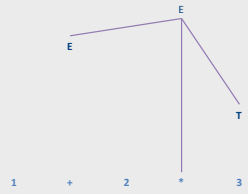
1 + 2 * 3

68

Intuition: Top-down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

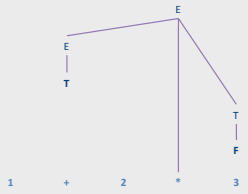


69

Intuition: Top-down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

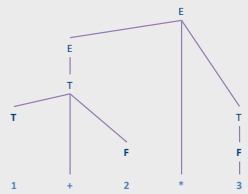


70

Intuition: Top-down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

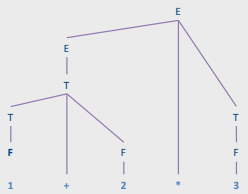


71

Intuition: Top-Down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

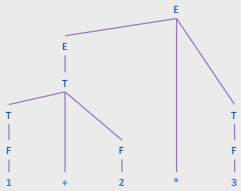


72

Intuition: Top-Down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)



73

Intuition: Bottom-Up Parsing

- Begin with the user's program
- Guess parse (sub)trees
- Check if root is the start symbol

74

Bottom-up parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

1 + 2 * 3

75

Bottom-up parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

F
1 + 2 * 3

76

Top-down parsing

Unambiguous grammar
E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

77

Top-down parsing

Unambiguous grammar
E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

78

Top-down parsing

Unambiguous grammar
E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

79

Bottom-up parsing

Unambiguous grammar
E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

80

Top-down parsing

Unambiguous grammar

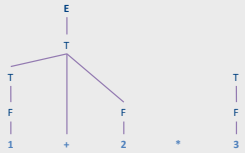
E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

81

Top-down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)

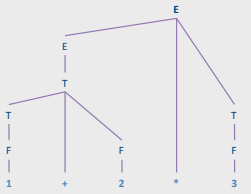


82

Top-down parsing

Unambiguous grammar

E → E * T
E → T
T → T + F
T → F
F → id
F → num
F → (E)



83

Challenges in top-down parsing

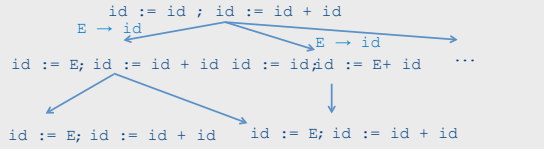
- Top-down parsing begins with virtually no
- information
  - Begins with just the start symbol, which matches *every program*
- How can we know which productions to apply?

84

"Brute-force" Parsing

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

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



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

Challenges in top-down parsing

- Top-down parsing begins with virtually no
- information
  - Begins with just the start symbol, which matches *every program*
- How can we know which productions to apply?
- In general, we can't
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong
- If we have to guess, how do we do it?
  - Parsing as a search algorithm
  - Too expensive in theory (exponential worst-case time) and practice

Predictive parsing

- Given a grammar G and a word w attempt to 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

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

not (not true or false)

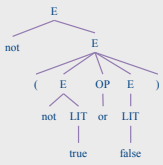
Boolean expressions example

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

not (not true or false)

production to apply known from next token

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)



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

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

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

- Variable **`current`** holds the current input token

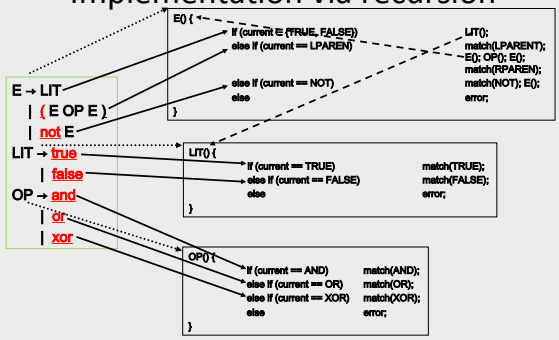
Functions for nonterminals

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

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

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

Implementation via recursion



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

Problem 1: productions with common prefix

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]`

Problem 2: null productions

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"?

p. 127

Problem 3: left recursion

E → E - term | term

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

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

97

FIRST sets

- For every production rule A→α
  - FIRST(α) = all terminals that α can start with
  - 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

98

Computing FIRST sets

- Assume no null productions A → ε
  1. Initially, for all nonterminals A, set FIRST(A) = { t | A → tω *for some ω* }
  2. Repeat the following until no changes occur:
     for each nonterminal A
       for each production A → Bω
         set FIRST(A) = FIRST(A) ∪ FIRST(B)
- This is known as fixed-point computation

99

FIRST sets computation example

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

STMT	EXPR	TERM

100

1. Initialization

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant

101

2. Iterate 1

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant
zero? Not ++ --		

102

2. Iterate 2

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant
zero? Not ++ --	id constant	

103

2. Iterate 3 – fixed-point

STMT → if EXPR then STMT
| while EXPR do STMT
| EXPR ;
EXPR → TERM -> id
| zero? TERM
| not EXPR
| ++ id
| -- id
TERM → id
| constant

STMT	EXPR	TERM
if while	zero? Not ++ --	id constant
zero? Not ++ --	id constant	
id constant		

104

p. 189

FOLLOW sets

- What do we do with nullable (ε) productions?
  - A→B C D B→ε C→ε
  - 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))

105

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)
  - For every two productions A→α and A→β we have FIRST(α) ∩ FIRST(β) = {}
    and FIRST(A) ∩ FOLLOW(A) = {}
- A language is said to be LL(k) when it has an LL(k) grammar

106

Back to problem 1

term → ID | indexed_elem
indexed_elem → ID [expr]

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

107

Solution: 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)

108

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

109

Back to problem 2

S → A a b
A → a | ε

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

- FIRST/FOLLOW conflict

110

Solution: 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

111

Back to problem 3

E → E - term | term

- Left recursion cannot be handled with a bounded lookahead
- What can we do?

112

p. 130

Left recursion removal

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

G_1 G_2

- $L(G_1) = \beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots$
- $L(G_2)$ = same

Can be done algorithmically. Problem: grammar becomes mangled beyond recognition

- For our 3rd example:

$E \rightarrow E$ - term | term ⇒ $E \rightarrow$ term TE | term, $TE \rightarrow$ - term TE | ε

113

LL(k) Parsers

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

114

LL(k) parsing via 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

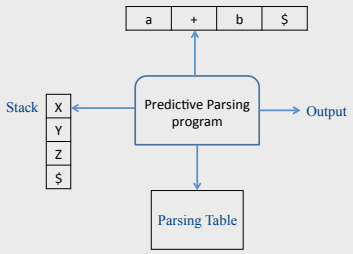
115

LL(k) parsing via 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

116

Model of non-recursive predictive parser



117

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	

118

Running parser 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

119

Erorrs

120

Handling Syntax Errors

- Report and locate the error
- Diagnose the error
- Correct the error
- Recover from the error in order to discover more errors
  - without reporting too many "strange" errors

121

Error Diagnosis

- Line number
  - may be far from the actual error
- The current token
- The expected tokens
- Parser configuration

122

Error Recovery

- Becomes less important in interactive environments
- Example heuristics:
  - Search for a semi-column and ignore the statement
  - Try to "replace" tokens for common errors
  - Refrain from reporting 3 subsequent errors
- Globally optimal solutions
  - For every input w, find a valid program w' with a "minimal-distance" from w

123

Illegal input 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

124

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 `b S` anyway "missing token b inserted in line XXX"

	a	b	c
S	S → a c	S → b S	

125

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 → b S	

126

Error handling and recovery

- x = a * (p+q * (-b * (r-s);
  - Where should we report the error?
  - The valid prefix property

127

The Valid Prefix Property

- For every prefix tokens
  - $t_1, t_2, \ldots, t_i$ that the parser identifies as legal:
    - there exists tokens $t_{i+1}, t_{i+2}, \ldots, t_n$ such that $t_1, t_2, \ldots, t_n$ is a syntactically valid program
- If every token is considered as single character:
  - For every prefix word u that the parser identifies as legal there exists w such that u.w is a valid program

Bring any memories?

128

Recovery is tricky

- Heuristics for dropping tokens, skipping to semicolon, etc.

129

Building the Parse Tree

130

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

131

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(LPAREN));
    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;
}
```

132

Parser for Fully Parenthesized Expers

```
static int Parse_Expression(Expression **expr_p) {
  Expression *expr = *expr_p = new_expression() ;
  /* try to parse a digit */
  if (Token.class == DIGIT) {
        expr->type='D';  expr->value=Token.repr -'0';
        get_next_token();
        return 1;       }
/* try parse parenthesized expression */
if (Token.class == '(') {
      expr->type='P';   get_next_token();
      if (!Parse_Expression(&expr->left))  Error("missing expression");
      if (!Parse_Operator(&expr->oper))  Error("missing operator");
      if (Token.class != ')') Error("missing )");
      get_next_token();
      return 1; }
return 0;
}
```


Famous last words

Syntax Analysis (Parsing)

- input
  - Sequence of tokens
- output
  - Abstract Syntax Tree
- Report syntax errors
  - unbalanced parenthesizes
- Create "symbol-table"
- Create pretty-printed version of the program

Why use context free grammars for defining PL syntax?

- Captures program structure (hierarchy)
- Employ formal theory results
- Automatically create "efficient" parsers

Capabilities and limitations of CFGs

- CFGs naturally express
  - Hierarchical structure
    - A program is a list of classes,
      A Class is a list of definition,
      A definition is either…
  - Beginning-end type of constraints
    - Balanced parentheses S → (S)S | ε
- Cannot express
  - Correlations between unbounded strings (identifiers)
  - Variables are declared before use: ω S ω
  - Handled by semantic analysis

p. 173

137

See you next week

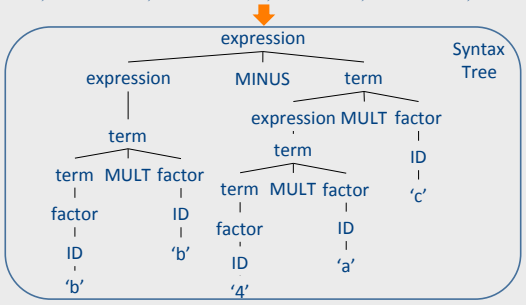
- Dan David 001

138

139

Parsers: from tokens to AST

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



140

Conventions

- a, b, … are input symbols.
  - But sometimes we allow ε as a possible value.
- …, X, Y, Z are stack symbols.
- …, w, x, y, z are strings of input symbols.
- α, β,… are strings of stack symbols.

"dangling-else" example

p. 174

Ambiguous grammar

S → if E then S
S | if E then S else S
| other

Unambiguous grammar

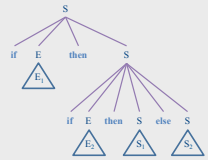
S → M | U
M → if E then M else M
| other
U → if E then S
| if E then M else U

Matched statements

Unmatched statements

if E_1 then if E_2 then S_1 else S_2

if E_1 then (if E_2 then S_1 else S_2)



if E_1 then (if E_2 then S_1) else S_2