Lecture 03 – Syntax analysis: top-down parsing

THEORY OF COMPILATION

Eran Yahav
You are here

Compiler

Source text

Lexical Analysis
Syntax Analysis
Semantic Analysis
Inter. Rep. (IR)
Code Gen.

Executable code

txt

exe
Last Week: from characters to tokens

\[ x = b^2 - 4ac \]

Token Stream

```xml
<ID,"x"> <EQ> <ID,"b"> <MUL> <ID,"b"> <MINUS> <INT,4> <MUL> <ID,"a"> <MUL> <ID,"c"> 
```
## Last Week: Regular Expressions

<table>
<thead>
<tr>
<th>Basic Patterns</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>The character x</td>
</tr>
<tr>
<td>.</td>
<td>Any character, usually except a new line</td>
</tr>
<tr>
<td>[xyz]</td>
<td>Any of the characters x,y,z</td>
</tr>
</tbody>
</table>

### Repetition Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R?</td>
<td>An R or nothing (=optionally an R)</td>
</tr>
<tr>
<td>R*</td>
<td>Zero or more occurrences of R</td>
</tr>
<tr>
<td>R+</td>
<td>One or more occurrences of R</td>
</tr>
</tbody>
</table>

### Composition Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1R2</td>
<td>An R1 followed by R2</td>
</tr>
<tr>
<td>R1</td>
<td>R2</td>
</tr>
</tbody>
</table>

### Grouping

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R)</td>
<td>R itself</td>
</tr>
</tbody>
</table>
Today: from tokens to AST

\[
\begin{align*}
<\text{ID},"x"> & \ <\text{EQ}> & \ <\text{ID},"b"> & \ <\text{MULT}> & \ <\text{ID},"b"> & \ <\text{MINUS}> & \ <\text{INT},4> & \ <\text{MULT}> & \ <\text{ID},"a"> & \ <\text{MULT}> & \ <\text{ID},"c"> \\
\end{align*}
\]

Syntax Tree
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

\[ S \rightarrow S;S \]
\[ S \rightarrow \text{id} := E \]
\[ E \rightarrow \text{id} \mid E + E \mid E \ast E \mid (E) \]

\[ V = \{S, E\} \]
\[ T = \{ \text{id}, '+', '*', '(', ')\} \]
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 + id
--------------------------  E → id
id := id ; id := id + id
--------------------------  E → 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

\[ x := y + z \times w \]

\[ S \rightarrow S;S \]
\[ S \rightarrow \text{id} := E \]
\[ E \rightarrow \text{id} | E + E | E \times E | (E) \]
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)
Leftmost Derivation

\[ x := z; \]
\[ y := x + z \]

\[ S \rightarrow S;S \]
\[ S \rightarrow \text{id} := \text{E} \]
\[ \text{E} \rightarrow \text{id} \mid \text{E} + \text{E} \mid \text{E} \ast \text{E} \mid (\text{E}) \]

\[
\begin{align*}
S & \rightarrow S;S \\
S & \rightarrow \text{id} := \text{E} \\
\text{E} & \rightarrow \text{id} \\
\text{E} & \rightarrow \text{id} := \text{E} \\
\text{E} & \rightarrow \text{id} := \text{E} + \text{E} \\
\text{E} & \rightarrow \text{id} := \text{id} + \text{E} \\
\text{E} & \rightarrow \text{id} := \text{id} + \text{id} \\
\text{x} := z & ; \text{y} := x + z
\end{align*}
\]
Rightmost Derivation

\[
x := z; \\
y := x + z
\]

\[
S \rightarrow S;S \\
S \rightarrow \text{id} := E \\
E \rightarrow \text{id} | E + E | E \times E | (E)
\]
Bottom-up Example

\[ x := z; \]
\[ y := x + z \]

\[ S \rightarrow S;S \]
\[ S \rightarrow \text{id} := E \]
\[ E \rightarrow \text{id} \mid E + E \mid E \ast E \mid (E) \]

\[
\begin{align*}
\text{id} & := \text{id} \; ; \; \text{id} := \text{id} + \text{id} \\
\text{id} & := E \; ; \; \text{id} := \text{id} + \text{id} \\
S & ; \; \text{id} := \text{id} + \text{id} \\
S & ; \; \text{id} := E + \text{id} \\
S & ; \; \text{id} := E + E \\
S & ; \; \text{id} := E \\
S & ; \; S \\
S & \rightarrow S;S
\end{align*}
\]

Bottom-up picking left alternative on every step \(\rightarrow\) Rightmost derivation when going top-down

17
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

\[
x := z; \\
y := x + z
\]

\[
S \rightarrow S ; S \\
S \rightarrow \text{id} := E \\
E \rightarrow \text{id} | E + E | E * E | (E)
\]

id := id ; id := id + id

E → id

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

...
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)
  
  already read...  to be read...

- Bottom-up (shift reduce)
Grammar Hierarchy

Non-ambiguous CFG

CLR(1)

LALR(1)

LL(1)

SLR(1)

LR(0)
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 =>
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

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

- Define a function for every nonterminal
- Every function works 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

\[ E \rightarrow \text{LIT} \mid (E \text{ OP } E) \mid \text{not } E \]
\[ \text{LIT} \rightarrow \text{true} \mid \text{false} \]
\[ \text{OP} \rightarrow \text{and} \mid \text{or} \mid \text{xor} \]

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

$E \to LIT$
| $(E\ OP\ E)$
| $not\ E$

$LIT \to$ $true$
| $false$

$OP \to$ $and$
| $or$
| $xor$

void $E()$
{
    if (current $\in \{TRUE, FALSE\})$
        LIT();
    else if (current == LPARENT)
        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;
}
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
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

```c
void A() {
    choose an A-production, A -> X_1X_2...X_k;
    for (i=1; i<= k; i++) {
        if (X_i is a nonterminal)
            call procedure X_i();
        elseif (X_i == 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?

\[
\text{term} \rightarrow \text{ID | indexed\_elem} \\
\text{indexed\_elem} \rightarrow \text{ID [ expr ]}
\]

- The function for indexed\_elem will never be tried...
  - What happens for input of the form
    - ID [ expr ]
Recursive descent: are we done?

```
S() {
    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?

\[ E \rightarrow E - \text{term} \]

```c
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. \( \text{term} \rightarrow \text{ID} \mid \text{indexed}_\text{elem} \)
   \( \text{indexed}_\text{elem} \rightarrow \text{ID} \ [ \text{expr} \] \)

2. \( S \rightarrow A \ a \ b \)
   \( A \rightarrow a \mid \varepsilon \)

3. \( E \rightarrow E \ - \ \text{term} \)

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

- For every production rule $A \rightarrow \alpha$
  - $\text{FIRST}(\alpha) =$ all terminals that $\alpha$ can start with
  - i.e., every token that can appear as first in $\alpha$ under some derivation for $\alpha$

- In our Boolean expressions example
  - $\text{FIRST}(\text{LIT}) = \{ \text{true, false} \}$
  - $\text{FIRST}( ( E \text{ OP } E ) ) = \{ '{' \}$
  - $\text{FIRST}( \text{ not } E ) = \{ \text{not} \}$

- No intersection between FIRST sets $\Rightarrow$ can always pick a single rule

- If the FIRST sets intersect, may need longer lookahead
  - $\text{LL}(k) =$ class of grammars in which production rule can be determined using a lookahead of $k$ tokens
  - $\text{LL}(1)$ 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 $A \rightarrow \alpha$
  - $\text{FOLLOW}(A) =$ 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
  - $\text{FIRST}(A_k) \cup (\text{if } A_k \text{ is nullable then } \text{FOLLOW}(N))$
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 1\textsuperscript{st} example

\begin{quote}
\begin{verbatim}
term → ID | indexed_elem
indexed_elem→ ID [ expr ]
\end{verbatim}
\end{quote}

- FIRST(ID) = \{ ID \}
- FIRST(indexed_elem) = \{ ID \}
- FIRST/FIRST conflict
Left factoring

- Rewrite the grammar to be in LL(1)

\[
\begin{align*}
term & \rightarrow ID \mid indexed\_elem \\
indexed\_elem & \rightarrow ID [ expr ]
\end{align*}
\]

\[
\begin{align*}
term & \rightarrow ID \ after\_ID \\
after\_ID & \rightarrow [ expr ] \mid \epsilon
\end{align*}
\]

Intuition: just like factoring $x*y + x*z$ into $x*(y+z)$
Left factoring – another example

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S \\
\quad \mid \text{if } E \text{ then } S \\
\quad \mid T \\
\]

\[
S \rightarrow \text{if } E \text{ then } S \ S' \\
\quad \mid T \\
S' \rightarrow \text{else } S \mid \varepsilon
\]
Back to our 2nd example

\[
\begin{align*}
S &\rightarrow A \ a \ b \\
A &\rightarrow a \mid \varepsilon
\end{align*}
\]

- FIRST(S) = \{ ‘a’ \}, FOLLOW(S) = \{
- FIRST(A) = \{ ‘a’ \varepsilon \}, FOLLOW(A) = \{ ‘a’ \}
- FIRST/FOLLOW conflict
Substitution

\[
S \rightarrow A \ a \ b \\
A \rightarrow a \mid \varepsilon
\]

Substitute A in S

\[
S \rightarrow a \ a \ b \mid a \ b
\]

Left factoring

\[
S \rightarrow a \ after\_A \\
after\_A \rightarrow a \ b \mid b
\]
Back to our 3rd example

\[ E \rightarrow E \text{ - term} \]

- Left recursion cannot be handled with a bounded lookahead

- What can we do?
Left recursion removal

\[ L(G_1) = \beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots \]

\[ L(G_2) = \text{same} \]

- For our 3\textsuperscript{rd} example:

\[ E \rightarrow \text{term} \]

\[ E \rightarrow \text{term} \text{TE} \]

\[ \text{TE} \rightarrow - \text{term} \text{TE} | \epsilon \]
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

<table>
<thead>
<tr>
<th>Nonterminals</th>
<th>(</th>
<th>)</th>
<th>not</th>
<th>true</th>
<th>false</th>
<th>and</th>
<th>or</th>
<th>xor</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Which rule should be used
Simple Example

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 |

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>aAb</td>
<td>A → c</td>
</tr>
</tbody>
</table>
### Simple Example

**Input suffix** | **Stack content** | **Move**
--- | --- | ---
abcbb$ | A$ | predict(A,a) = A $\rightarrow$ aAb
abcbb$ | aAb$ | match(a,a)
bcbb$ | Ab$ | predict(A,b) = ERROR

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A $\rightarrow$ aAb</td>
<td>A $\rightarrow$ c</td>
</tr>
</tbody>
</table>
Error Handling

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

\[ x = a \times (p+q \times (-b \times (r-s))) \]

- Where should we report the error?
- The valid prefix property
- Recovery is tricky
  - Heuristics for dropping tokens, skipping to semicolon, etc.
Error Handling in LL Parsers

\[ S \rightarrow \text{a c} \mid \text{b S} \]

<table>
<thead>
<tr>
<th>Input suffix</th>
<th>Stack content</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>c$</td>
<td>S$</td>
<td>predict(S,c) = ERROR</td>
</tr>
</tbody>
</table>

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

Result: infinite loop

<table>
<thead>
<tr>
<th>Input suffix</th>
<th>Stack content</th>
<th>Move</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc$</td>
<td>S$</td>
<td>predict(b,c) = S \rightarrow bS</td>
</tr>
<tr>
<td>bc$</td>
<td>bS$</td>
<td>match(b,b)</td>
</tr>
<tr>
<td>c$</td>
<td>S$</td>
<td>Looks familiar?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S → a c</td>
<td>S → bS</td>
<td></td>
</tr>
</tbody>
</table>
Error Handling

- Requires more systematic treatment
- Enrichment
  - Acceptable-set method
  - Not part of course material
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
Coming up next time

- More syntax analysis