City University London

IN2009

Language Processors

Week 3

Parsing I (syntax analysis)

Igor Siveroni

Session Plan

Session 3: Parsing (syntax analysis)

- Syntax definition
  - context free grammars (BNF)
- Parse trees
- Ambiguous grammars
- Removal of left recursion
- Recursive descent parsing

RegExp with Abbreviations

- It is useful to introduce abbreviations to regular expressions i.e. to use intermediate names. For example:

integer = 0 | [1-9] [0-9]+

can be defined as:

integer = 0 | [1-9] digit+
digit = [0-9]

Syntax definition

- We need to recognise structures like expressions with parentheses, or nested statements:
  - (109+23) (1+(250+3))
  - **`if (…) then if (…)`** stms… **`else`** … **`else`** …
- How do we do this?

Syntax definition

- It is tempting to use regular expressions with abbreviations

```
digits  =  [0-9]+
sum     =  expr "+" expr
expr    =  "(" sum ")" | digits
```

Syntax definition

- But remember that regular expression abbreviations like digits work like macros, and are *substituted* directly to the original definition, so we would get

expr = "(" sum ")" | digits

expr = "(" expr "+" expr ")" | digits

expr = "(" “(" (expr "+" expr) ")" | digits) "+" expr) ")" | digits

...

When do we stop? We can’t use recursive definitions with Regular Expressions

Syntax definition

- An automaton cannot be created from such definitions.
- What we need is a notation where recursion does not mean abbreviation and substitution, but instead means ***definition***...
- Recursion gives additional expressive power.

Syntax definition

- Then, (1+(250+3)) can be recognised by our recursive definitions

expr →	"(" sum ")"	
→	"(" expr "+" expr ")"	(using the sum definition)
→	"(" digits "+" expr ")"	(using the expr definition)
→	"(" 1 "+" expr ")"	
→	"(" 1 "+" "(" sum ")" ")"	
→	"(" 1 "+" "(" expr "+" expr ")" ")"	
→	"(" 1 "+" "(" digits "+" expr ")" ")"	
→	"(" 1 "+" "(" digits "+" digits ")" ")"	
→	"(" 1 "+" "(" 250 "+" digits ")" ")"	
→	"(" 1 "+" "(" 250 "+" 3 ")" ")"	

Syntax definition

- Alternation within definitions is then not needed, since
  - **r = ab(c|d)e** is the same as:
    r = abne
    n = c
    n = d
    so alternation not needed at all!
  - we will however retain alternation at the top level of definition.

Syntax definition

- repetition via Kleene closure * is not needed, since

  e= (abc)* can be expressed by

  e = (abc)e

  e = ε

- this recursive notation is called *context-free grammars* or BNF (see Session 1)
  - recognised by pushdown automata (PDA); recognition is implemented in many ways
  - involves (implicitly or explicitly) building the concrete syntax (parse) tree, matching against the tokens produced by the lexical analyser
  - once again, a tool can produce a parser for us

Context-free grammars

- A *language* is a set of *strings*
- Each string is a finite sequence of *symbols* taken from a finite *alphabet*
- For parsing: symbols = lexical tokens, alphabet = set of token types returned by the lexical analyser
- A grammar describes a language
- A grammar has a set of *productions* of the form

  symbol → *symbol symbol* … *symbol*

- Zero or more symbols on RHS
- Each symbol either a *terminal* from the alphabet or a *non-terminal* (appears on LHS of some productions)
- No token ever on LHS of production
- One non-terminal distinguished as *start symbol* of the grammar

Syntax for straight-line programs

1	$S \rightarrow S\ ;\ S$
2	$S \rightarrow$ id := E
3	$S \rightarrow$ print (L)
4	$E \rightarrow$ id
5	$E \rightarrow$ num
6	$E \rightarrow E + E$
7	$E \rightarrow$ (S , E)
8	$L \rightarrow E$
9	$L \rightarrow L\ ,\ E$

- a *context-free grammar*
- terminal symbols (tokens):

  id print num , ( ) := ; +
- non-terminal symbols:

  S E L
- start symbol S

Derivations

```
a := 7;
b := c + (d:= 5+6, d)
```

Repeatedly replace any non-terminal by one of its right-hand sides.

Leftmost derivation: Always replace the leftmost non-terminal.

Rightmost derivation: Always replace the rightmost non-terminal.

$\underline{S}$

S ; $\underline{S}$

$\underline{S}$; id := E

id := $\underline{E}$; id := E

id := num ; id := $\underline{E}$

id := num ; id := E + $\underline{E}$

id := num ; id := $\underline{E}$ + (S , E)

id := num ; id := id + ($\underline{S}$, E)

id := num ; id := id + (id := $\underline{E}$, E)

id := num ; id := id + (id := E + E , $\underline{E}$)

id := num ; id := id + (id := $\underline{E}$+E, id)

id := num ; id := id + (id := num+$\underline{E}$, id)

id := num ; id := id + (id := num + num, id)

Concrete syntax derivations and parse trees

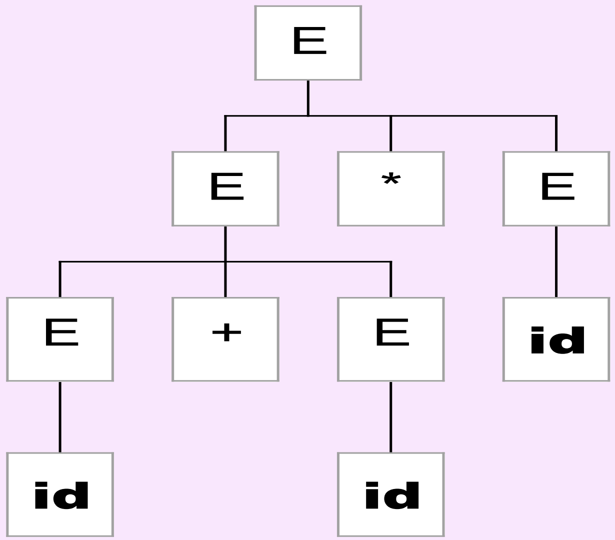
E → E * E | E / E | E **+** E | E **-** E | **(** E **)** | **id** | **num**

Leftmost derivation of **id + id * id** :

Derivation:

E
→ E * E
→ E **+** E * E
→ **id +** E * E
→ **id + id** * E
→ **id + id * Id**

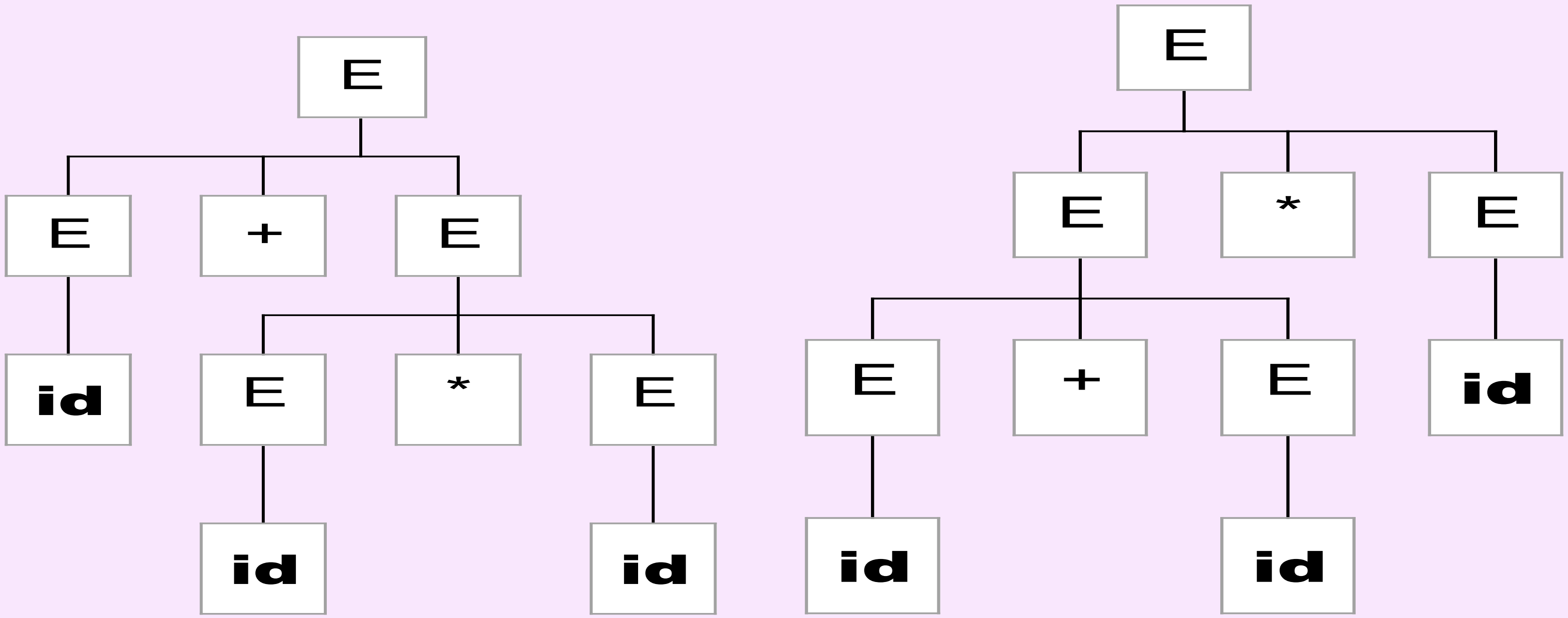
Concrete syntax tree (parse tree) built from derivation:



Ambiguous grammars

E → E * E | E / E | E **+** E | E **-** E | **(** E **)** | **id** | **num**

Two parse trees for **id + id * id**

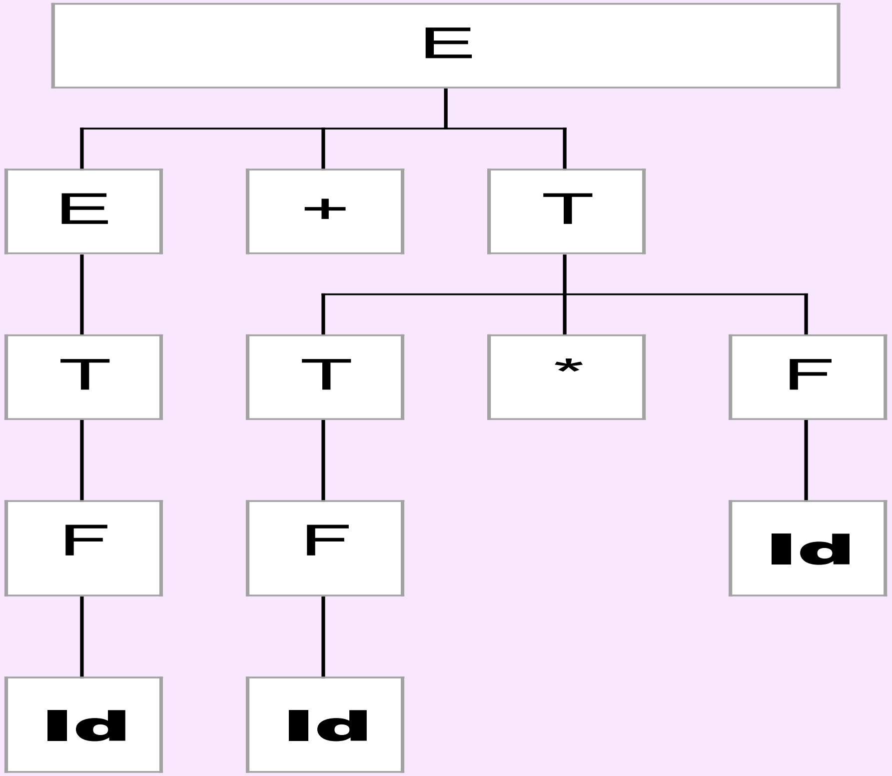


A grammar is **ambiguous** if there exists a string that can be represented by two different parse trees.

Disambiguating the grammar

E → E + T
E → E - T
E → T
T → T * F
T → T / F
T → F

F → id
F → num
F → (E)



Only one tree now possible from this BNF for input string

id + id * id

Recursive descent parsing

- AKA “Top down” / Predictive
  - One recursive function per non-terminal
  - Each grammar production turns into one clause of a recursive function
  - Only works on grammars where the **first** terminal symbol of each grammatical construct <u>provides enough information to choose the production</u>

Recursive descent parsing

S → if E then S else S
S → begin S L
S → print E
L → end
L → ; S L
E → num = num

```
void eat(int t) {
 if (tok==t) advance();
 else  error();
}

void advance() {
  tok = getToken(); }
```

```
void S() {
   switch (tok) {
   case IF:
        eat(IF); E(); eat(THEN);
        S(); eat(ELSE); S(); break;
   case BEGIN: eat(BEGIN); S(); L();
   break;
   case PRINT: eat(PRINT); E();  break;
}

void E() {
    eat(NUM); eat(EQ); eat(NUM);
}
```

Appel 2002, (p46, Gram 3.11)

But...

- if we try to implement a recursive descent parser for the disambiguated expression grammar…

$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$

$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow F$

$F \rightarrow id$
$F \rightarrow num$
$F \rightarrow (E)$

```
void E() {
        switch (tok) {
                case ???: E(); eat(PLUS); T(); break;
                case ???: E(); eat(MINUS); T(): break;
                case ???: T(); break;
        }}
```

- Problems:
  - **no initial terminal symbol** to tell us which production to choose
  - Even if we compute the FIRST sets, more then one production to choose from (due to left-recursion)
  - Even worse…there's a potential infinite loop!!

Eliminating left recursion

$X \rightarrow X\,\gamma$
$X \rightarrow \alpha$

can always be rewritten

$X \rightarrow \alpha\, X'$
$X' \rightarrow \gamma\, X'$
$X' \rightarrow$

$S \rightarrow E\ \$$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$
$E' \rightarrow -\ T\ E'$
$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$
$T' \rightarrow /\ F\ T'$
$T' \rightarrow$

$F \rightarrow \mathrm{id}$
$F \rightarrow \mathrm{num}$
$F \rightarrow (E)$

Sketch of resulting recursive descent parser

$S \rightarrow E\ \$$

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow -\ T\ E'$

$E' \rightarrow$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

$T' \rightarrow /\ F\ T'$

$T' \rightarrow$

$F \rightarrow \text{id}$

$F \rightarrow \text{num}$

$F \rightarrow (E)$

```
void E() { T(); E'(); }

void E'() {
        switch (tok) {
                case PLUS: eat(PLUS); T(); E'(); break;
                case MINUS: eat(MINUS); T(); E'(); break;
                default: /* empty - that's ok */ break;
        }
}
```

JavaCC:parser & lexical analysis

- Fortunately, we don’t have to hand-code parsers…
- Given an (E)BNF grammar, software tools like JavaCC will produce a parser for us.
- Parser input: Tokens, defined by regular expressions (lexical specification).
- Lexical analyser also generated by JavaCC.

CourseWork

- Coursework
  - OUT: Wednesday 10 February
  - DUE: Wednesday 24 February. 8pm.
- Test: Regular Expressions
  - Monday 22 February
  - A217/A218
  - 1-2pm. We'll finish class early.