NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 6: Top-down parsing and LL(1) parser construction

2020-01-24

Michael Engel

Includes material by Jan Christian Meyer

Overview

- Ambiguity of grammars revisited
- Elimination of left recursion
- Top-down parsing
    - Recursive descent parsers: structure and implementation
    - Table-driven LL(1) parsers
        - Table generation

Ambiguity of grammars

- For the compiler, it is important that each sentence in the language defined by a context-free grammar has a ***unique*** rightmost (or leftmost) ***derivation***

- A grammar in which multiple rightmost (or leftmost) derivations exist for a sentence is called an ***ambiguous grammar***
  - it can produce multiple derivations and multiple parse trees

- Multiple parse trees imply ***multiple possible meanings for a single program!*** ⚡

Ambiguity of grammars: example

"***dangling else***"-problem in ALGOL-like languages (e.g. PASCAL)

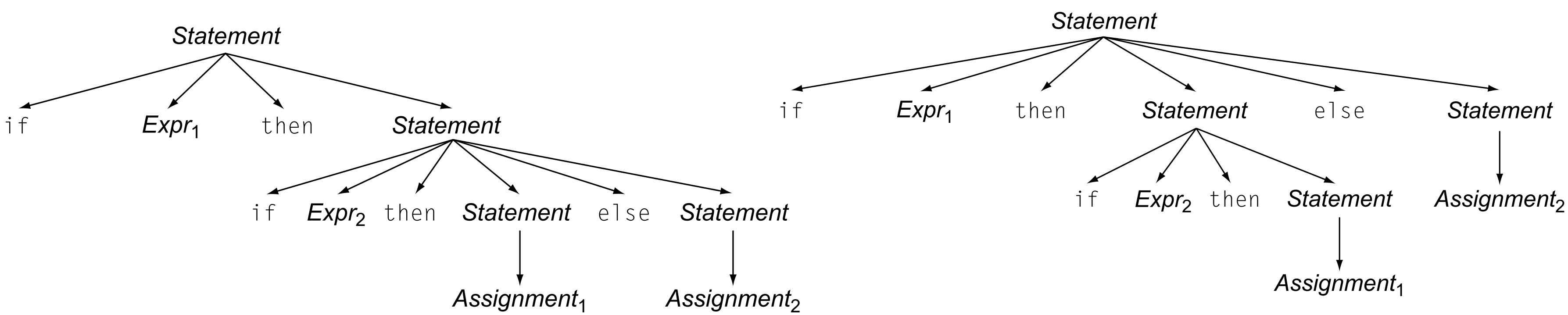
```
Statement → if Expr then Statement else Statement
          | if Expr then Statement
          | Assignment
          | …other statements…
```

"else" part is optional

This statement

```
if Expr1 then if Expr2 then Assignment1 else Assignment2
```

has two distinct rightmost derivations with different behaviors:



Removing ambiguity

We can modify the grammar to include a rule that determined which **if** controls an **else**:

```
Statement → if Expr then Statement
          | if Expr then WithElse else Statement
          | Assignment
WithElse  → if Expr then WithElse else WithElse
          | Assignment
```

This solution restricts the set of statements that can occur in the **then** part of an **if-then-else** construct

- It ***accepts the same set of sentences*** as the original grammar
- but ensures that each else has an unambiguous match to a specific if

Removing ambiguity: example

The modified grammar has only one rightmost derivation for the example

```
1  Statement → if Expr then Statement
2             | if Expr then WithElse else Statement
3             | Assignment
4  WithElse  → if Expr then WithElse else WithElse
5             | Assignment
```

```
if Expr1 then if Expr2 then Assignment1 else Assignment2
```

Rule	Sentential form
	Statement
1	**if** *Expr* **then** *Statement*
2	**if** *Expr* **then** **if** *Expr* **then** *WithElse* **else** *Statement*
3	**if** *Expr* **then** **if** *Expr* **then** *WithElse* **else** *Assignment*
5	**if** *Expr* **then** **if** *Expr* **then** *Assignment* **else** *Assignment*

Order of derivations

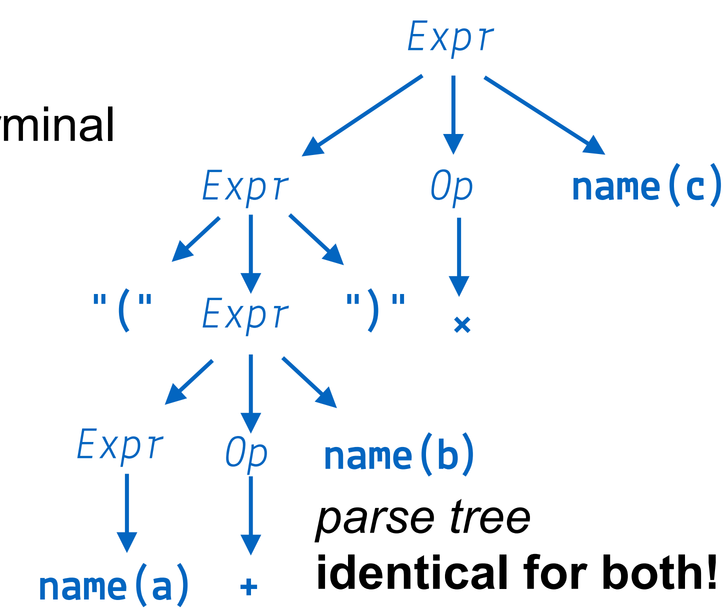

```
1  Expr → "(" Expr ")"
2       | Expr Op name
3       | name
4  Op   → +
5       | -
6       | ×
7       | ÷
```

Rightmost:
rewrite, at each step, the rightmost nonterminal

Rule	Sentential form
	Expr
2	*Expr* *Op* **name**
6	*Expr* × **name**
1	"(" *Expr* ")" × **name**
2	"(" *Expr* *Op* **name** ")" × **name**
4	"(" *Expr* + **name** ")" × **name**
3	"(" **name** + **name** ")" × **name**

Leftmost: rewrite, at each step, the leftmost nonterminal

Rule	Sentential form
	Expr
2	*Expr* *Op* **name**
1	"(" *Expr* ")" *Op* **name**
2	"(" *Expr* *Op* **name** ")" *Op* **name**
3	"(" **name** *Op* **name** ")" *Op* **name**
4	"(" **name** + **name** ")" *Op* **name**
6	"(" **name** + **name** ")" × **name**



parse tree
identical for both!

Left factoring

- Parsers (and scanners) only have a limited ***lookahead*** to upcoming tokens

- Example: given a production

  $$A \rightarrow \mathbf{abcdef}\ X\ \mathbf{gh}\ |\ \mathbf{abcdef}\ Y\ \mathbf{gh}$$

  the parser is unable to choose between the two production if it can only look one character ahead
- As with NFA→DFA conversion, if we can postpone the decision until it makes a difference, that works
  - Rewriting the grammar as

    $A \rightarrow \mathbf{abcdef}\ A'$

    $A' \rightarrow X\ \mathbf{gh}\ |\ Y\ \mathbf{gh}$

    preserves the language by adding one production to collect a common prefix shared by several other productions

Left recursion

Syntax analysis

- Let’s consider this grammar for a list of 'a’s:

  $A \rightarrow A\mathbf{a} \mid \mathbf{a}$

  which derives the following words:

  $A \rightarrow \mathbf{a}$

  $A \rightarrow A\mathbf{a} \rightarrow \mathbf{aa}$

  $A \rightarrow A\mathbf{a} \rightarrow A\mathbf{aa} \rightarrow \mathbf{aaa}$

  …

- The production $A \rightarrow A\mathbf{a}$ is ***left recursive***, the head (nonterminal symbol) always appears on the left side of the production

An equivalent grammar

- The same sequences can be generated by this grammar:

$$A \rightarrow \mathbf{a}A'$$
$$A' \rightarrow \mathbf{a}A' \mid \varepsilon$$

the empty string ε returns from the production

It derives the following words:

$A \rightarrow \mathbf{a}$

$A \rightarrow \mathbf{a}A' \rightarrow \mathbf{aa}A' \rightarrow \mathbf{aa}$

$A \rightarrow \mathbf{a}A' \rightarrow \mathbf{aa}A' \rightarrow \mathbf{aaa}A' \rightarrow \mathbf{aaa}$

…

Eliminating left recursion

- If a nonterminal has *m* productions that are left recursive and *n* productions that are not

  $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_m$

  $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \mid \beta_n$

  greek letters (except ε) stand for arbitrary combinations of other (non-)terminals

  we can introduce A' and rewrite the productions as (see [1]):

  $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots \mid \beta_n A'$

  $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

- This generates the same language and removes (immediate) left recursion
  - "Immediate" because left recursion can also happen in several steps (indirectly), e.g. in the following productions

    $A \rightarrow B\mathbf{x}$ and $B \rightarrow A\mathbf{y}$ result in $A \rightarrow B\mathbf{x} \rightarrow A\mathbf{yx}$

    Here, A again shows up on the left when derived from A

What can we do with CFGs now?

Syntax analysis

- So far, we have encountered (see also [2])
  - Context-Free Grammars, their derivations and syntax trees
  - Ambiguous grammars, and mentioned that there’s no single, true way to disambiguate them (it depends on what we want them to stand for)
  - Left factoring, which always shortens the distance to the next nonterminal
  - Left recursion elimination, which always shifts a nonterminal to the right

Recursive descent parsing

- Example: grammar that models "if" and "while" statements:

  P → **if** *COND* **then** *STATEMENT* **end**
  | **if** *COND* **then** *STATEMENT* **else** *STATEMENT* **end**
  | **while** *COND* **do** *STATEMENT* **end**

- Let's make it a bit simpler:

  P → **i***C***t***S***z** | **i***C***t***S***e***S***z** | **w***C***d***S***z**
  C → **c**
  S → **s**

- Let us parse the string "**ictsesz**"
- A top-down parser begins at the ***start symbol*** *P* and chooses a production:

P
↓
???

Recursive descent: what next?

Syntax analysis

- If we can only look ahead by one token and read an "**i**", we can choose between two productions:

$$P \rightarrow \mathbf{i}C\mathbf{t}S\mathbf{z}$$
$$\quad | \ \mathbf{i}C\mathbf{t}S\mathbf{e}S\mathbf{z}$$

- We cannot make this choice before seeing more of the token stream
- Left factoring makes this problem decidable with only one character of lookahead
- It generates the following grammar:

$$P \rightarrow \mathbf{i}C\mathbf{t}SP' \ | \ \mathbf{w}C\mathbf{d}S\mathbf{z}$$
$$P' \rightarrow \mathbf{z} \ | \ \mathbf{e}S\mathbf{z}$$
$$C \rightarrow \mathbf{c}$$
$$S \rightarrow \mathbf{s}$$

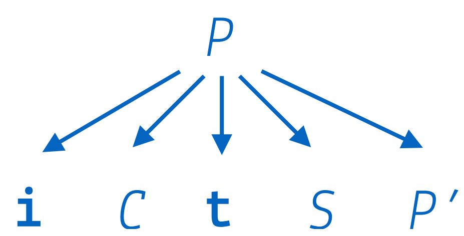
Recursive descent: what next?

$$
\begin{aligned}
P &\rightarrow \mathtt{i}C\mathtt{t}SP' \mid \mathtt{w}C\mathtt{d}S\mathtt{z} \\
P' &\rightarrow \mathtt{z} \mid \mathtt{e}S\mathtt{z} \\
C &\rightarrow \mathtt{c} \\
S &\rightarrow \mathtt{s}
\end{aligned}
$$

- Now we only have one production to choose from when reading an "i":

$$P \rightarrow \mathtt{i}C\mathtt{t}SP'$$

- and we can generate the ***parse tree*** equivalent to the derivation:



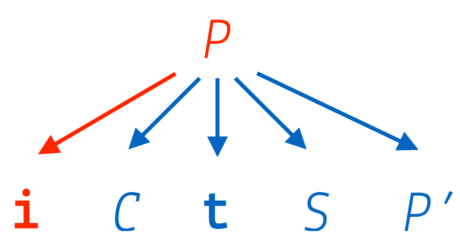
Recursive descent: going down…

Syntax analysis

- Recursive descent implies that we follow the children of the current parse tree node down to the leaves (which must be terminal symbols)

$$P \rightarrow \mathtt{i}C\mathtt{t}SP' \mid \mathtt{w}C\mathtt{d}S\mathtt{z}$$
$$P' \rightarrow \mathtt{z} \mid \mathtt{e}S\mathtt{z}$$
$$C \rightarrow \mathtt{c}$$
$$S \rightarrow \mathtt{s}$$

- So let's see if we can parse "**ictsesz**"
- We follow the tree from *P* to its first child:



The input token sequence:

`ictsesz`

↑

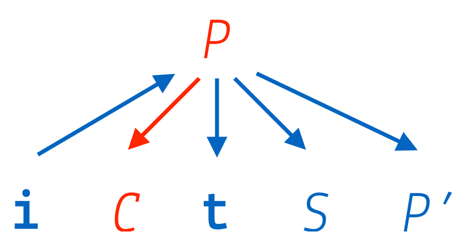
the arrow indicates the parser's position in the token stream

- we have an "i" as lookahead
  ⇒ ***matches** the first production for P!*
- Now, the remaining token stream is "**ctsesz**"

Backtrack and repeat

$P \rightarrow$ `i`C`t`SP' | `w`C`d`S`z`
$P' \rightarrow$ `z` | `e`S`z`
$C \rightarrow$ `c`
$S \rightarrow$ `s`

- we have an "i" as lookahead ⇒ ***match!***
- Now, the remaining token stream is **"ctsesz"**
  - We return (backtrack) to P to continue parsing:



The input token sequence:

`i ctsesz`

↑

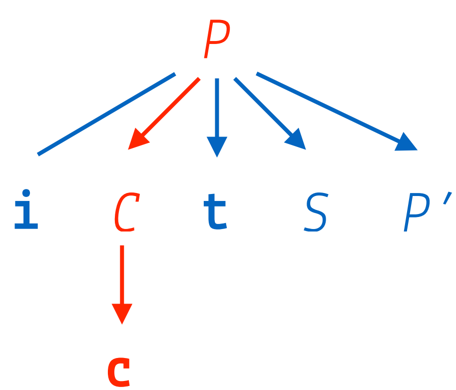
- This gives us the nonterminal C
- A nonterminal cannot match any token, so we need to pick another production

Pick the next production

Syntax analysis

- There is only one choice to expand *C*
  - When going from *P* to *C* in the previous step, we did not consume a token
- The lookahead is now **c**
  - Pick production **C → c** and expand the tree:

$P \rightarrow$ **i***C***t***SP′* | **w***C***d***S***z**
$P′ \rightarrow$ **z** | **e***S***z**
$C \rightarrow$ **c**
$S \rightarrow$ **s**



The input token sequence:

i ctsesz

↑

- we have a "c" as lookahead ⇒ "**tsesz**"

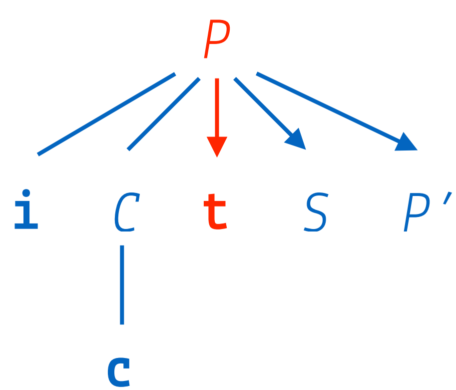
The next terminal symbol

Syntax analysis

- The next terminal symbol in *P* is **t**

- The lookahead is also **t**
  - Consume the token and expand the tree once more:

P	→	**i***C***t***SP′* | **w***C***d***S***z**
P′	→	**z** | **e***S***z**
C	→	**c**
S	→	**s**



The input token sequence:

ic tsesz

↑

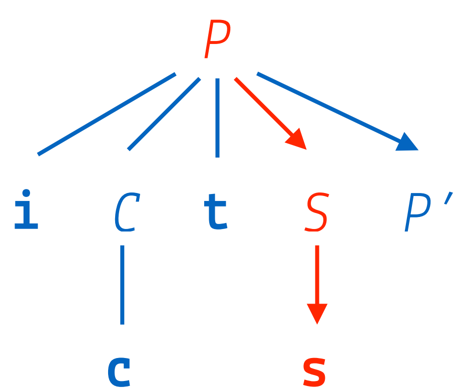
- remaining token stream:"**sesz**"

The next nonterminal symbol S

Syntax analysis

- The next nonterminal in the first production is *S*, so we apply its production
- The lookahead is now **s**
  - This matches the pattern derived from *S*, so we can expand the tree again:

P → **i***C***t***SP′* | **w***C***d***S***z**
P′ → **z** | **e***S***z**
C → **c**
S → **s**



The input token sequence:

ict sesz

↑

- remaining token stream: "**esz**"

The next nonterminal symbol S

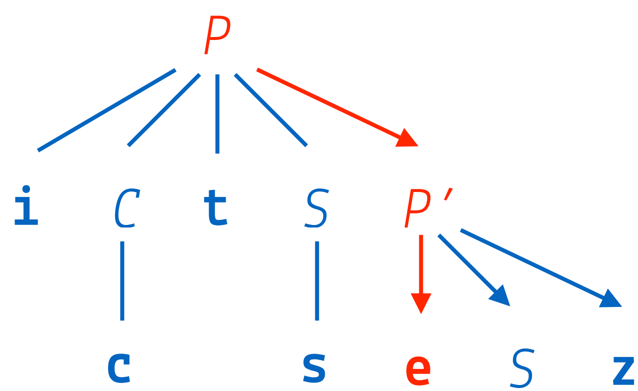
Syntax analysis

- The final nonterminal in the first production is P'
- Now we have to choose between:

  $P' \rightarrow$ **z** and $P' \rightarrow$ **e**S**z**

  We can now choose the right production using only one token of lookahead!

$P \rightarrow$ **i**C**t**SP' | **w**C**d**S**z**
$P' \rightarrow$ **z** | **e**S**z**
$C \rightarrow$ **c**
$S \rightarrow$ **s**



The input token sequence:

icts esz

↑

- remaining token stream: **"sz"**

The final steps

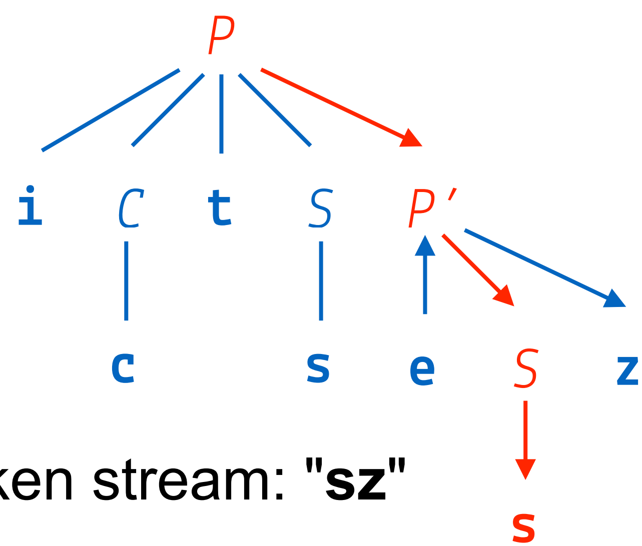
Syntax analysis

- The remaining steps are similar to ones we have already seen
- Take the next nonterminal symbol

  S and match the input to production $S \rightarrow$ **s**

  We can again choose the right production using only one symbol of lookahead!

$P \rightarrow$ **i**C**t**SP' | **w**C**d**S**z**
$P' \rightarrow$ **z** | **e**S**z**
$C \rightarrow$ **c**
$S \rightarrow$ **s**



The input token sequence:

ictse sz

↑

- remaining token stream: "**sz**"

Validated!

Syntax analysis

- The remaining nonterminal in the production $P' \rightarrow$ **e**S**z** is **z**
- This matches the remaining input token

  → **we backtrack and find no further children**

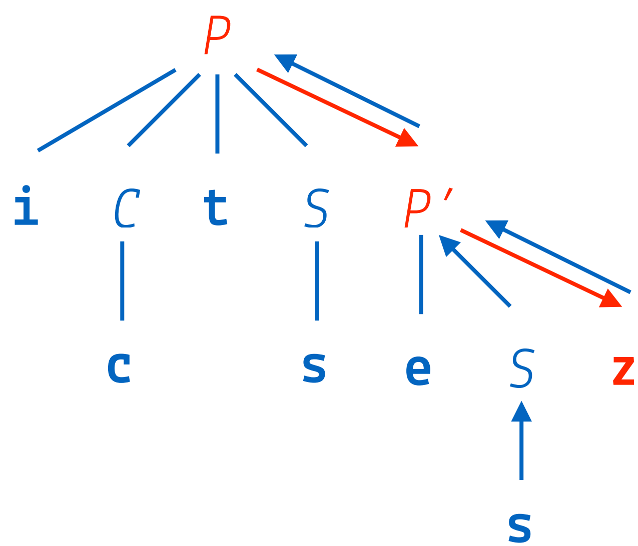
  → **we we able to match all characters, thus the input matches our grammar**

$P \rightarrow$ **i**C**t**SP' | **w**C**d**S**z**

$P' \rightarrow$ **z** | **e**S**z**

$C \rightarrow$ **c**

$S \rightarrow$ **s**



The input token sequence:

ictses z → ictsesz ✔

Top-down parsing summarized

Syntax analysis

- Predictive parsing by recursive descent:
  - Start from the start symbol (top)
  - Verify terminals
  - Pick a unique production for nonterminals based on the lookahead
  - Expand the syntax tree by productions and recursively treat the new subtree in the same way
- This requires that the grammar is suitable, but we can adapt them somewhat
  - Left factor where a common lookahead prevents picking the right production
  - Eliminate left-recursive productions
  - We only saw left factoring in action so far, but let's do one other grammar

LL(1) parsing:

- **scan from Left to right**
- **use Leftmost derivation**
- **1 symbol lookahead**

Implementing recursive descent

Syntax analysis

- Recursive descent parsers can easily be implemented by hand
- Example: parsing A = **a**A**c** | **b**

- We can naively try to implement the parser like this:

```
symbol sym;
…
sym = next();
if (sym == 'a') {
    sym = next();
    if (sym == A)   { sym = next(); } else { error(); }
    if (sym == 'c') { sym = next(); } else { error(); }
}
else
if (sym == 'b')   { sym = next(); } else { error(); }
```

A = **a**A**c**
|
b

next() is the interface to the scanner!

Wait… will this work?

Correct implementation

- Example: parsing *A* = **a***A***c** | **b**
- Whenever we encounter a ***nonterminal*** such as *A* we have to parse its ***production***!
- Let us implement the parser as a function:

What is the correct way to call A() from main to ensure the parser works correctly in all cases?

```
symbol sym;
…
void A(void) {
  if (sym == 'a') {
     sym = next();
     A();
     if (sym == 'c') { sym = next(); } else { error(); }
  }
  else
  if (sym == 'b')    { sym = next(); } else { error(); }
}
```

A = **a***A***c** | **b**

Recursively calling the parser for A allows to parse arbitrarily nested inputs!

Some more implementation hints (not in C) can be found in [3]

Table-driven parsing

- As with scanners, coding a recursive descent parser for a complex language is lots of work and error prone
- Idea: use tables to configure the parser
  - parser makes decisions based on indexing (nonterminal, terminal) pairs and find a single production

- To make that table, it's a good idea to determine
  - What can the strings derived from a nonterminal begin with?
  - Which nonterminals can vanish, so that the lookahead symbol is actually part of the *next* production to choose?
  - What can come directly after a nonterminal that can vanish?

(where 'vanish' means that there is a production $X \rightarrow \varepsilon$, so that nonterminal X disappears from the intermediate form in the derivation without consuming any characters from the input token stream)

Another example grammar

$$
\begin{aligned}
S &\rightarrow \mathbf{u}\ B\ D\ \mathbf{z} \\
B &\rightarrow B\ \mathbf{v} \mid \mathbf{w} \\
D &\rightarrow E\ F \\
E &\rightarrow \mathbf{y} \mid \varepsilon \\
F &\rightarrow \mathbf{x} \mid \varepsilon
\end{aligned}
$$

It doesn't model anything in particular, it's just a useful example

FIRST

Syntax analysis

$S \to$ **u** B D **z**
$B \to B$ **v** | **w**
$D \to E\ F$
$E \to$ **y** | **ε**
$F \to$ **x** | **ε**

- The set FIRST(α) is the set of terminals that can appear to the left in α
  - α is any combination of terminals and nonterminals
- If we tabulate FIRST for all the heads in the grammar, we obtain
  - FIRST(S) = {**u**} – **u** begins the only production
  - FIRST(B) = {**w**} – however many times $B \to B$**v** is taken, **w** appears on the left in the end
  - FIRST(E) = {**y**} – only production that derives any terminal
  - FIRST(F) = {**x**} – ditto
  - FIRST(D) = {**y**,**x**}
    - **y** because $D \to EF \to$ **y**F
    - **x** because $D \to EF \to F \to$ **x** (E can disappear by $E \to$ **ε**)

FOLLOW

Syntax analysis

$S \rightarrow$ **u** B D **z**
$B \rightarrow B$ **v** | **w**
$D \rightarrow E$ F
$E \rightarrow$ **y** | ε
$F \rightarrow$ **x** | ε

- FOLLOW (*N*) for a nonterminal *N* is the set of terminals that can appear directly to its right
  - In order to find these, you have to examine all the places *N* appears in production bodies, and find the terminals directly to its right
  - If it has a nonterminal on its right, you have to follow all its productions too, and find out what can come up instead of it
    - That will be its FIRST set
  - If it has a nonterminal that can vanish to its right, you have to look at what comes afterwards...
    - ...and in general, collect all the terminals that can appear to the right in one way or another
- This is a little trickier than FIRST, but it can be done manually
  - See fig. 3.8, p. 106 in [4] for an algorithm to compute FOLLOW

FOLLOW for our grammar

Syntax analysis

$S \rightarrow$ **u** B D **z**
$B \rightarrow B$ **v** | **w**
$D \rightarrow E\ F$
$E \rightarrow$ **y** | ε
$F \rightarrow$ **x** | ε

- FOLLOW(S) = {**$**} (the end of input)
- FOLLOW(B) = {**v**, **x**, **y**, **z**} taken from the derivations
  - $S \rightarrow$ **u**BD**z** $\rightarrow$ **u**B**v**D**z**
  - $S \rightarrow$ **u**BD**z** $\rightarrow$ **u**BEF**z** $\rightarrow$ **u**BF**z** $\rightarrow$ **u**B**xz**
  - $S \rightarrow$ **u**BD**z** $\rightarrow$ **u**BEF**z** $\rightarrow$ **u**B**y**F**z**
  - $S \rightarrow$ **u**BD**z** $\rightarrow$ **u**BEF**z** $\rightarrow$ **u**BF**z** $\rightarrow$ **u**B**z**
- FOLLOW(D) = {**z**} (from $S \rightarrow$ **u**BD**z**)
- FOLLOW(E) = {**x**, **z**} taken from the derivations
  - $S \rightarrow$ **u**BD**z** $\rightarrow$ **u**BEF**z** $\rightarrow$ **u**BE**xz**
  - $S \rightarrow$ **u**BD**z** $\rightarrow$ **u**BEF**z** $\rightarrow$ **u**BE**z**
- FOLLOW(F) = {**z**} – from $S \rightarrow$ **u**BD**z** $\rightarrow$ **u**BEF**z**

Nullability

$S \to$ **u** B D **z**
$B \to B$ **v** | **w**
$D \to E\ F$
$E \to$ **y** | ε
$F \to$ **x** | ε

Syntax analysis

- A nonterminal is ***nullable*** if it can produce the empty string (in any number of steps)
  - Here, the notation might be different between various textbooks
  - E.g., the Aho/Ullman/Seti/Lam "Dragon book" [5] (one of the standard compiler textbooks) denotes this by putting **ε** in the FIRST set
  - We denote it by keeping a separate record
- To summarize,
  - nullable (S) = no – there are terminals in the only production
  - nullable (B) = no – there are terminals in both productions
  - nullable (E) = yes – it produces $E \to \varepsilon$
  - nullable (F) = yes – it produces $F \to \varepsilon$
  - nullable (D) = yes – $D \to EF \to F \to \varepsilon$

Building the parsing table

Syntax analysis

- Obtain the FIRST and FOLLOW sets and nullable information for your grammar

- Consider every production $X \rightarrow \alpha$ in the grammar, and apply two rules

  - Enter the production $X \rightarrow \alpha$ at (X,`t`) where `t` is in FIRST(α)

  - When $\alpha \rightarrow^{*} \varepsilon$, enter the production $X \rightarrow \alpha$ at (X,`t`) where `t` is in FOLLOW(X)

Oops, a left recursion!

Syntax analysis

This will not work, expanding B on lookahead 'w' requires a choice the parser cannot make

	u	w	v	x	y	z
S	*S*→**u***BD***z**					
B		*B*→**w** *B*→*B***v**				
D				*D*→*EF*	*D*→*EF*	
E					*E*→**y**	
F				*F*→**x**		

Fix the grammar

- Eliminating left recursion gives us

$S \rightarrow \mathbf{u}\ B\ D\ \mathbf{z}$
$B \rightarrow B\ \mathbf{v} \mid \mathbf{w}$
$D \rightarrow E\ F$
$E \rightarrow \mathbf{y} \mid \varepsilon$
$F \rightarrow \mathbf{x} \mid \varepsilon$

⇒

$S \rightarrow \mathbf{u}\ B\ D\ \mathbf{z}$
$B \rightarrow \mathbf{w}\ B'$
$B' \rightarrow \mathbf{v}\ B' \mid \varepsilon$
$D \rightarrow E\ F$
$E \rightarrow \mathbf{y} \mid \varepsilon$
$F \rightarrow \mathbf{x} \mid \varepsilon$

- Update the FIRST, FOLLOW, nullable sets after the change:
  - FIRST(B) = {**w**}, FOLLOW(B) = {**x**, **y**, **z**}, nullable(B) = no
  - FIRST(B') = {**v**}, FOLLOW(B') = {**x**, **y**, **z**}, nullable(B') = yes

This is better… after rule 1

Syntax analysis

	u	w	v	x	y	z
S	$S \rightarrow$ **u**BD**z**					
B		$B \rightarrow$ **w**B'				
B'			$B' \rightarrow$ **v**B'			
D				$D \rightarrow EF$	$D \rightarrow EF$	
E					$E \rightarrow$ **y**	
F				$F \rightarrow$ **x**		

Now apply rule 2

Where nonterminal symbols are ***nullable***, insert at FOLLOW

	u	w	v	x	y	z
S	$S \rightarrow$ **u**BD**z**					
B		$B \rightarrow$ **w**B'				
B'			$B' \rightarrow$ **v**B'	$B' \rightarrow \varepsilon$	$B' \rightarrow \varepsilon$	$B' \rightarrow \varepsilon$
D				$D \rightarrow EF$	$D \rightarrow EF$	$D \rightarrow EF$
E				$E \rightarrow \varepsilon$	$E \rightarrow$ **y**	$E \rightarrow \varepsilon$
F				$F \rightarrow$ **x**		$F \rightarrow \varepsilon$

Result: a LL(1) parse table

- There is only one rule to choose from given a combination (NT, T) of a nonterminal and a terminal symbol
- Thus, the parse tree can be built deterministically by following the method from the first example
  - Pick productions for NTs by looking them up in the table
  - Encountering a combination without production ⇒ error
- The LL(1) parse table can, of course, also be constructed by an algorithm that processes (parses) the input grammar
  - See [4], fig. 3.12, p. 113
    (note: the book adds the set $FIRST^{+}$ to simplify notation)
- This is the first step to create a ***parser generator*** (also called ***compiler compiler***)

So far, so good…

- Most programming language constructs can be expressed in a backtrack-free grammar
- Predictive parsers for these are simple, compact, and efficient
  - They can be implemented in a number of ways, including hand-coded, recursive descent parsers and generated LL(1) parsers, either table driven or direct coded
- The primary drawback of top-down, predictive parsers lies in their inability to handle left recursion
  - Left-recursive grammars model the left-to-right associativity of expression operators in a more natural way than right-recursive grammars
- What lies ahead?
  - More parsing: bottom up – L**R**(1) parsers
  - These are the basis for many parser generators, e.g. yacc/bison

References

Syntax analysis

[1] A.V. Aho, S.C. Johnson, J.D. Ullman:
Deterministic parsing of ambiguous grammars
Communications of the ACM, August 1975, doi:10.1145/360933.360969

[2] D.J. Rosenkrantz, R.E. Stearns:
Properties of Deterministic Top Down Grammars
Information and Control. 17 (3): 226–256, 1970. doi:10.1016/s0019-9958(70)90446-8

[3] Niklaus Wirth:
Compiler Construction
Original version: Addison-Wesley 1996, ISBN 0-201-40353-6
Revised edition 2017 freely available at
https://inf.ethz.ch/personal/wirth/CompilerConstruction/index.html

– in this small book of a bit more than 100 pages, Wirth explains the design and implementation of a small compiler for a subset of the Oberon language. This book is rather implementation-oriented, so don't expect too much theoretical detail

[4] Keith Cooper and Linda Torczon:
Engineering a Compiler (second Edition)
ISBN 9780120884780 (hardcover), 9780080916613 (ebook)

[5] Alfred Aho, Monica S. Lam, Ravi Sethi, Jeffrey Ullman:
Compilers: Principles, Techniques, and Tools (second edition)
Addison-Wesley 2006, ISBN 978-0321486813