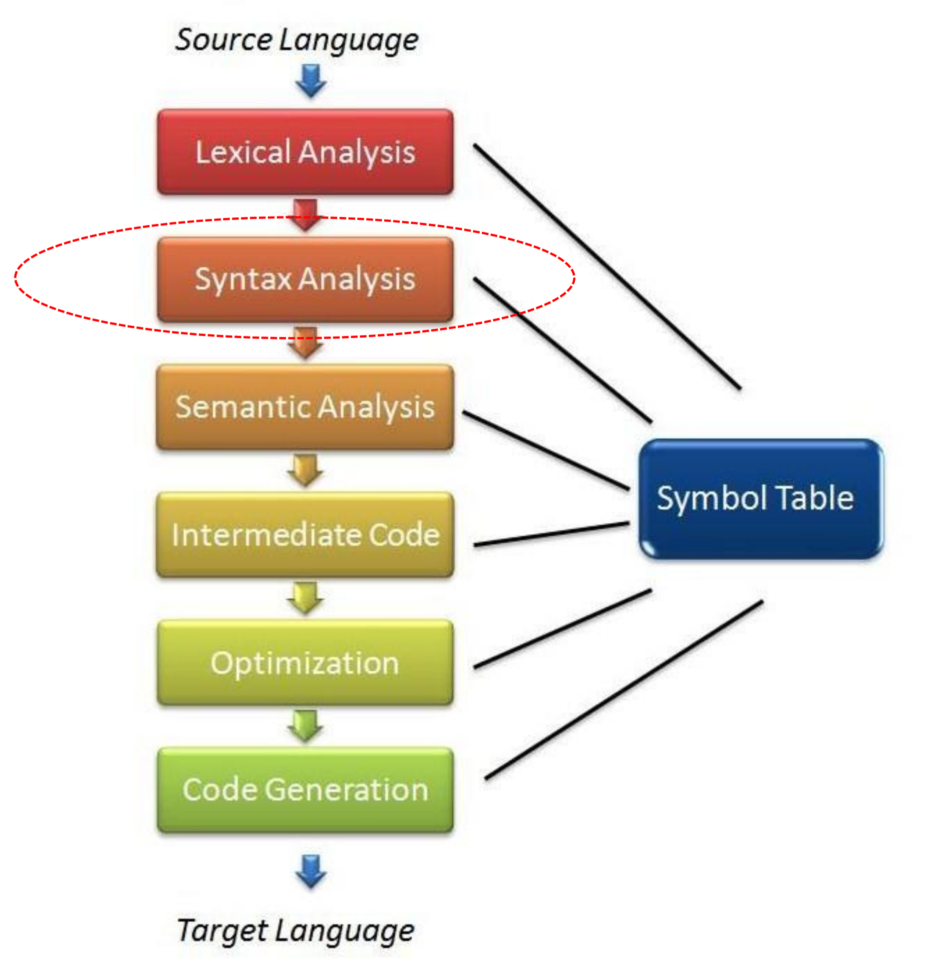
Compiler Construction

ARNOLD MEIJSTER
A.MEIJSTER@RUG.NL

Compiler Structure



(E)BNF

- BNF (Backus-Naur Form) is a meta-language used to describe the grammar of a programming language.

- (E)BNF stands for either (Extended) Backus-Naur Form or (Extended) Backus Normal Form.

- There are many dialects of (E)BNF in use, but the differences are almost always minor.

BNF

- < > indicate a *nonterminal* that needs to be further expanded, e.g. <variable>
- The symbol ::= means *is defined as*
- Symbols not enclosed in < > are *terminals;* they represent themselves, e.g. if, while, (
- The symbol | means *or;* it separates alternatives, e.g. <addop> ::= + | -
- This is *all there is* to "plain" BNF

Extended BNF (Backus Naur form)

- The following constructs are pretty standard:
  - [ ] enclose an optional part of the rule
    - Example:

```
<if statement> ::=
      if ( <condition> ) <statement> [ else <statement> ]
```

  - { } enclose a part that can be repeated zero or more times
    - Example:

```
<parameter list> ::= ( [<parameter> { , <parameter> }] )
```

Syntax analysis: Parser

- Parsing is the process of determining whether a string of tokens can be generated by a grammar.

- Most parsing methods fall into one of two classes, called the *top-down* and *bottom-up* methods.

- Efficient top-down parsers are easy to build by hand.
  - And can also be generated using automatic tools (e.g. LLnextgen)

- Bottom-up parsing, however, can handle a larger class of grammars.
  - And are almost always generated using automatic tools (e.g. Yacc/Bison)

Parsing: Top-Down, Bottom-Up

- Given the grammar: E -> 0 | E + E
- And a string to parse: "0 + 0"

- Top-down parsing: derivation from start symbol E:
  - E -> E + E -> 0 + E -> 0 + 0
- Bottom-up parsing: group terminals into RHS of rules:
  - 0 + 0 <- E + 0 <- E + E <- E

- Usually, parsing is done on-the-fly while tokens are read:
  - Top-down:
    - After seeing 0, we don't yet know which rule to use;
    - After seeing 0 +, we can **expand** E to E + E
  - Bottom-up:
    - 0 can be **reduced** to E right away, without seeing +

Three “Equivalent” Grammars

G1: <*subexp*> `::=` **`a`** | **`b`** | **`c`** | <*subexp*> **`-`** <*subexp*>

G2: <*subexp*> `::=` <*var*> **`-`** <*subexp*> | <*var*>
<*var*> `::=` **`a`** | **`b`** | **`c`**

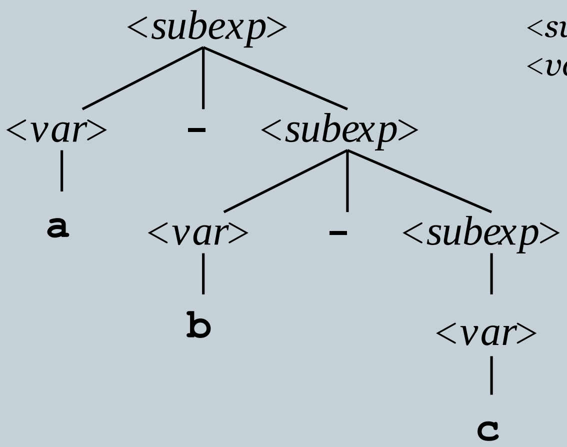
G3: <*subexp*> `::=` <*subexp*> **`-`** <*var*> | <*var*>
<*var*> `::=` **`a`** | **`b`** | **`c`**

These grammars all define the same language: the language of strings that contain one or more **a**s, **b**s or **c**s separated by minus signs. But...

They produce different parse trees for a-b-c

<subexp> `::=` *<var>* **-** *<subexp>* | *<var>*
<var> `::=` **a** | **b** | **c**

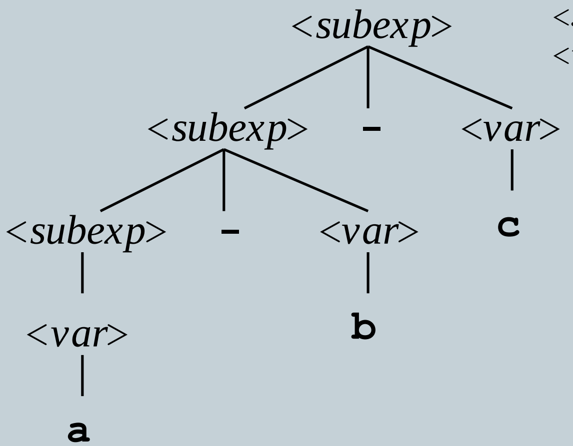
G2 parse tree:



This corresponds with a-(b-c), i.e. 5-(4-1)=2

<subexp> `::=` *<subexp>* - *<var>* | *<var>*
<var> `::=` **a** | **b** | **c**

G3 parse tree:



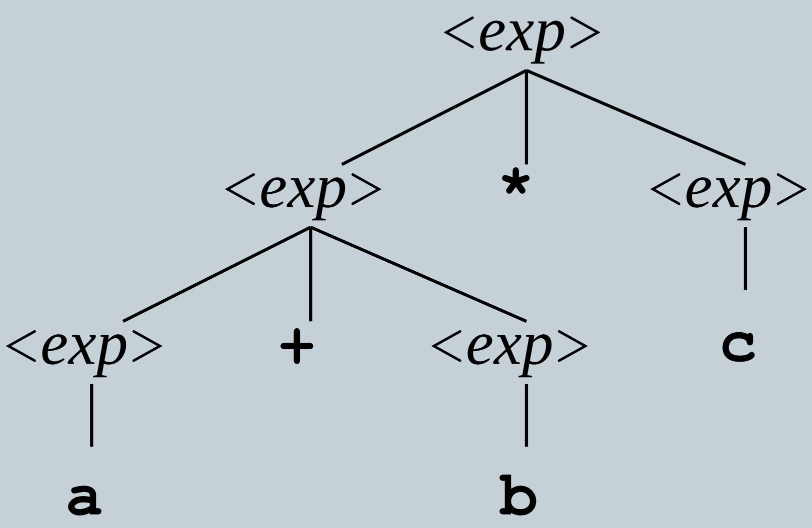
This corresponds with (a-b)-c, i.e. (5-4)-1=0

Why Parse Tree Structure Matters

- We want the structure of the parse tree to correspond to the semantics of the string it generates
- This makes grammar design much harder: we're interested in the structure of the parse tree, not just in the generated string
- Parse trees are where syntax meets semantics

Grammar for Expressions

<*exp*> `::=` <*exp*> `+` <*exp*>
| <*exp*> `*` <*exp*>
| `(`<*exp*>`)`
| `a` | `b` | `c`

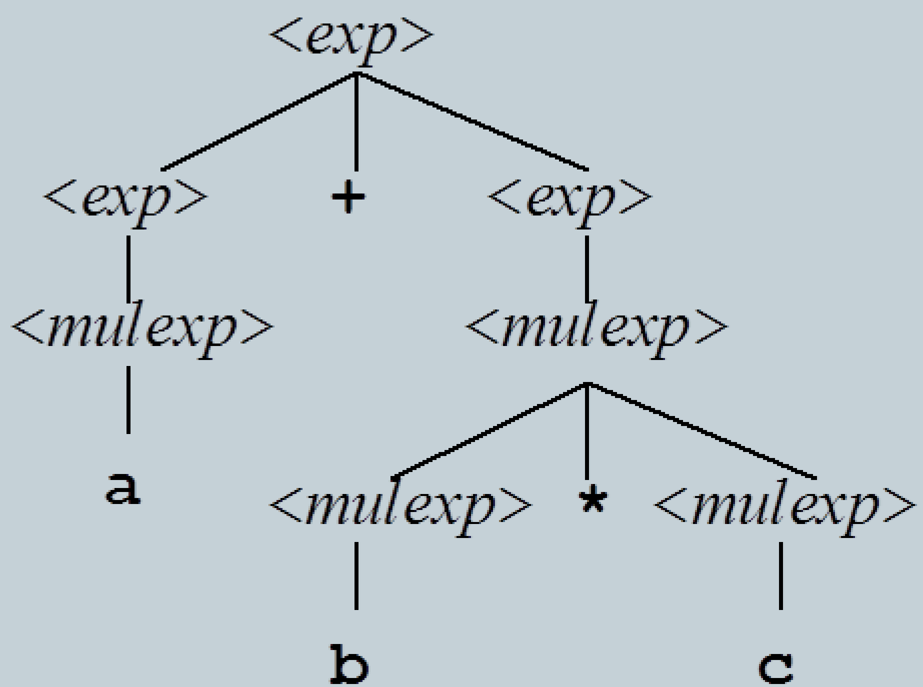


This is a *parse tree* for **`a+b*c`**.

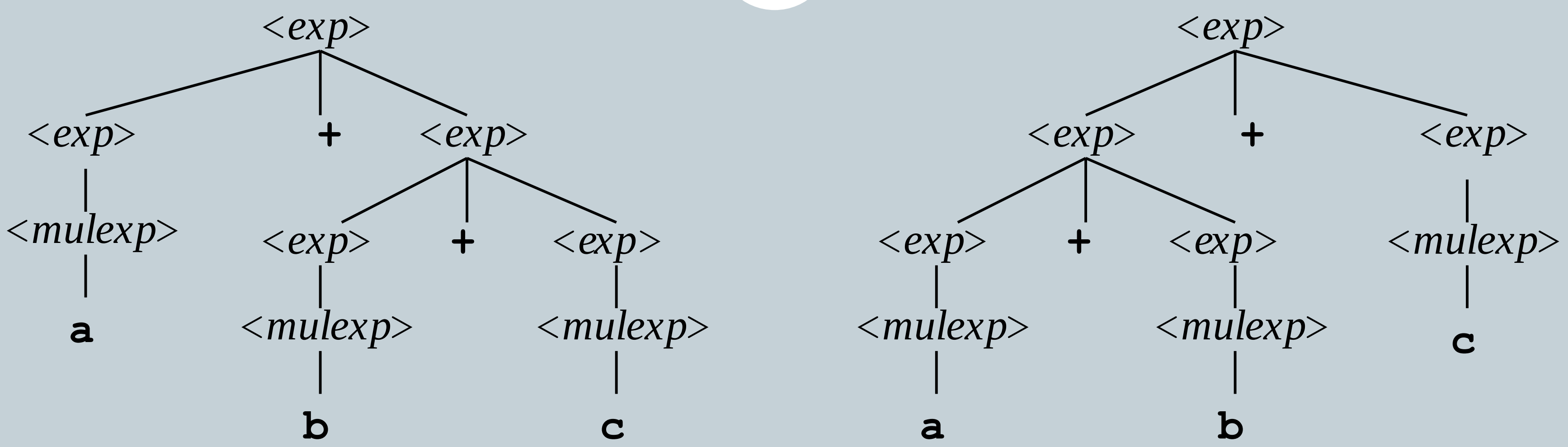
Problem: the addition is performed before the multiplication, which is not the usual convention for operator *precedence*.

Correct Precedence: a + b*c = a + (b*c)

```
<exp>     ::= <exp> + <exp> | <mulexp>
<mulexp>  ::= <mulexp> * <mulexp>
            | (<exp>)
            | a | b | c
```



We still have a problem: associativity



- The grammar for a language should generate a unique parse for every possible input.

- The grammar can generate either tree for **`a+b+c`**.

- The left one is not the usual convention for the *associativity* of +.

Associativity Examples

- In C/C++/Java:

  `a+b+c` — most operators are left-associative

  `a=b=0;` — right-associative (assignment)

- In Haskell:

  `3-2-1` — most operators are left-associative

  `1::2::[]` — right-associative (list construction)

- In Python:

  `a/b*c` — most operators are left-associative

  `a**b**c` — right-associative (exponentiation)

Associativity in the Grammar

- To fix the associativity problem, we modify the grammar to make parse trees of **+**s 'grow' to the left (and likewise for *****s)

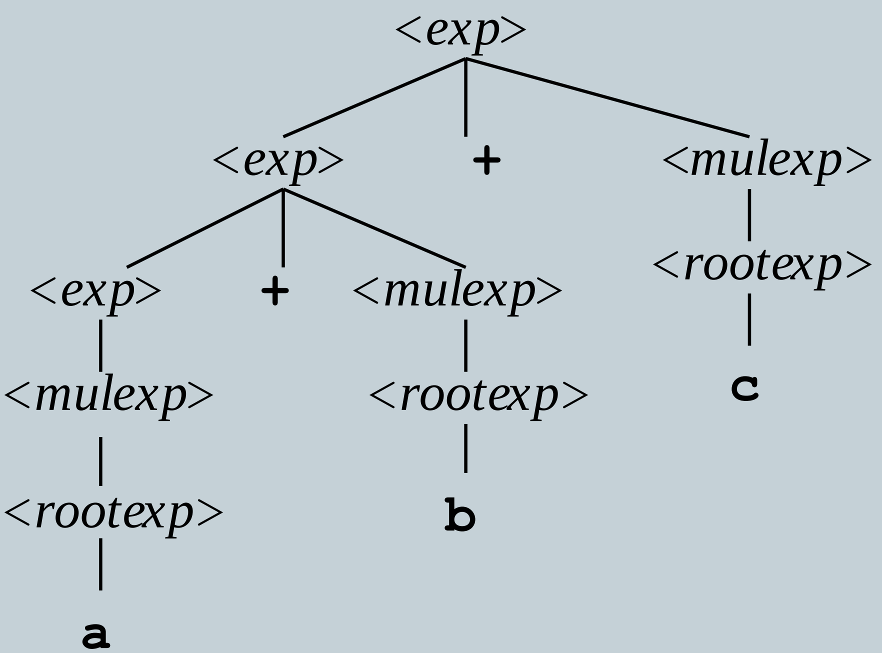
```
<exp>     ::= <exp> + <mulexp>        | <mulexp>
<mulexp>  ::= <mulexp> * <rootexp>    | <rootexp>
<rootexp> ::= (<exp>) | a | b | c
```

Correct parse tree for a + b + c

<exp> ::= *<exp>* **+** *<mulexp>* | *<mulexp>*

<mulexp> ::= *<mulexp>* ***** *<rootexp>* | *<rootexp>*

<rootexp> ::= **(***<exp>***)** | **a** | **b** | **c**



Grammars and Parsers

- LL(1) parsers
  - Top down
  - **L**eft-to-right input
  - **L**eftmost derivation
  - **1** symbol look-ahead

LL(1) grammars

- LR(1) parsers
  - Bottom up
  - **L**eft-to-right input
  - **R**ightmost derivation
  - **1** symbol look-ahead

LR(1) grammars

- Also: LL(k), LR(k), …

Parsing: Top-Down

- Generally:
  - top-down is easier to understand/implement directly
  - top-down parsing may require changes to the grammar

- Top-down parsing can be done:
  - Iteratively
  - Recursive descent
  - Table lookup and transitions

- A recursive descent parser does not require backtracking to take alternative paths along the parse (derivation) path.

Recursive-Descent Parsing (cont.)

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`

- Consider a BNF grammar rule (production) of the form

  <LHS> → <RHS>

- The coding process when there is only one RHS:
  - If the RHS starts with a terminal symbol, compare it with the next input token; if they match, continue, else there is an error
  - For a nonterminal symbol in the RHS, call its associated parsing subprogram

The routine `match()`

```
int match(int token) {
    if (nextToken != token) {
        return 0;  /* no match */
    }
    nextToken = lex();
    return 1;  /* match */
}
```

Recursive-Descent Parsing

A grammar for simple expressions:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | ( <expr> )
```

Recursive-Descent Parsing (cont.)

```
/* Function parseExpr
   Parses strings in the language generated by the rule:
   <expr> → <term> {(+ | -) <term>}
 */

void parseExpr() {
  parseTerm();
  while (match(PLUS_TOKEN) || match(MINUS_TOKEN)) {
    parseTerm();
  }
}
```

This routine does not detect errors!

Invariant: Each parsing routine leaves the next token in `nextToken`

Recursive-Descent Parsing (cont.)

```
/* Function parseTerm
   Parses strings in the language generated by the rule:
   <term> → <factor> {(* | /) <factor>}
 */

void parseTerm() {
  parseFactor();
  while (match(TIMES_TOKEN) || match(DIV_TOKEN)) {
    parseFactor();
  }
}
```

Again, this routine does not detect errors!

Recursive-Descent Parsing (cont.)

- A nonterminal that has more than one RHS requires a test to determine which RHS it is to parse:

  ```
  <factor> → id | ( <expr> )
  ```

  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error

Recursive-Descent Parsing (cont.)

```
/* Function parseFactor
   Parses strings in the language generated by the rule:
   <factor> -> id  |  (<expr>)  */

 void parseFactor() {
   if (match(IDENT)){
     /* skip, match() accepted the token */
     return;
   }
   if (match(LEFT_PAR)) {
     /* parse: (<expr>) */
     parseExpr();
     if (match(RIGHT_PAR)) {
       return;
     }
     syntaxError("Expected ')'");
   }
   syntaxError("Expected <identifier> or '('");
}
```

Note: the routine **`syntaxError`** aborts!

Left Recursion

- A grammar is ***left recursive*** if $\exists$ a non-terminal **A** such that $\mathbf{A} \Rightarrow^* \mathbf{A}\ \alpha$ without accepting a terminal.

What does $\Rightarrow^*$ mean?

A $\rightarrow$ B <u>x</u>
B $\rightarrow$ A <u>y</u>

Let α be any string of grammar symbols (i.e. terminals and non-terminals).

The notation $\alpha \Rightarrow^* \beta$ denotes that we can derive β starting from α in zero or more rewrites without accepting terminals/tokens (input).

Recursive-Descent Parsing (cont.)

The Left Recursion Problem

- If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser

- The parser will loop forever.
  - Often, a grammar can be modified to remove left recursion

Removing Left Recursion

- Two cases of left recursion:

#	Production rule
1	*<expr> → <expr> + <term>*
2	*| <expr> - <term>*
3	*| <term>*

#	Production rule
4	*<term> → <term> * <factor>*
5	*| <term> / <factor>*
6	*| <factor>*

- Transform as follows:

#	Production rule
1	*<expr> → <term> <expr2>*
2	*<expr2> → + <term> <expr2>*
3	*| - <term> <expr2>*
4	*| ε*

#	Production rule
4	*<term> → <factor> <term2>*
5	*<term2> → * <factor> <term2>*
6	*| / <factor> <term2>*
	| ε

LL Grammar Restriction 1

- **Grammar Restriction 1** (for top-down parsing):

An LL grammar contains no left-recursive rules.

Recursive-Descent Parsing (cont.)

- The other characteristic of grammars that disallows LL(1) top-down parsing is the inability to determine the correct RHS on the basis of one lookahead token
  - $A \rightarrow a \mid aB$

First Set

- **First(E)**, is the set of terminal symbols that may appear at the beginning of a sentence derived from E
  - And also includes ε if E can generate an empty string

- Def: $First(\alpha) = \{a \mid a\ is\ a\ terminal\ and\ \alpha \Rightarrow^{*} a\beta\}$
  - Note: If $\alpha \Rightarrow^{*} \epsilon$ then $\epsilon \in First(\alpha)$

- Example: E -> 0 | E + E
  - First(0) = {0}, First(E + E) = {0}, First(E) = {0} ∪ First(E + E) = {0}

First Set

Rules for computing First Sets

- If X is a terminal, then $First(X) = \{X\}$.
- If $X \Rightarrow^* \epsilon$, then place ϵ in $First(X)$.
- $First(X\alpha) = First(X)$ if $\epsilon \notin First(X)$.
- $First(X\alpha) = (First(X) \backslash \{\epsilon\}) \cup First(\alpha)$ if $\epsilon \in First(X)$.
- If X is a nonterminal, and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place s in $First(X)$ if s $\in First(Y_i)$, and $\epsilon \in First\left(y_j\right)$ for all $1 \leq j < i$.
  - If $\epsilon \in First(y_i)$ for all $1 \leq i \leq k$, then place ϵ in $First(X)$.

Example: Calculating FIRST Sets

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| - *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → `number`
11	| `identifier`

FIRST(rhs 3) = { + }
FIRST(rhs 4) = { – }
FIRST(rhs 5) = { ε }

FIRST(rhs 7) = { * }
FIRST(rhs 8) = { / }
FIRST(rhs 9) = { ε }

FIRST(rhs 10) = { `number`}

FIRST(rhs 11) = {`identifier` }

FIRST(factor) =FIRST(rhs 10) ∪ FIRST(rhs 11)
= { `number`, `identifier` }

FIRST(term2) =FIRST(rhs 7) ∪ FIRST(rhs 8) ∪FIRST(rhs 9)
= {*, /, ε }

FIRST(term) = FIRST(factor) = { `number`, `identifier` }

FIRST(expr2) = FIRST(rhs 3) ∪ FIRST(rhs 4) ∪FIRST(rhs 5)
= {+, -, ε }

FIRST(expr) = FIRST(term) = { `number`, `identifier` }

FIRST(goal)=FIRST(expr)= { `number`, `identifier` }

LL Grammar Restriction 2

- **Grammar Restriction 2** (for top-down parsing):

The First sets of all alternatives/choices for the same LHS must be different (so we know which path to take upon seeing the next terminal symbol/token).

Recursive-Descent Parsing (cont.)

- Pairwise Disjointness Test:
  - For each each pair of rules $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$ in the grammar, it must be true that

$$\mathrm{FIRST}(\alpha_i) \cap \mathrm{FIRST}(\alpha_j) = \phi$$

- Examples:

Disjoint: $A \rightarrow a \mid bB \mid cAb$

Overlap: $A \rightarrow a \mid aB$

Overlap: $A \rightarrow a \mid B \qquad B \rightarrow \{b\ c\}\ a$

Example Expression Grammar

<expr> → <term> { <op> <term> }
<term> → 'ident' | '(' <expr> ')'
<op> → '+' | '-'

```
void parseTerm ( ) {
  if (match(IDENT)) {
    return;
  }
  if (match(LPAR)) {
    parseExpr();
    if (match(RPAR)) {
      return;
    }
    syntaxError("Expected ')'");
    return;
  }
  syntaxError("Expected Ident or ')'");
}
```

```
void parseExpr ( ) {
  parseTerm();
  while (token∈First(op)){
      parseOp();
      parseTerm();
  }
}
```

Recursive-Descent Parsing (cont.)

Left factoring can resolve First set conflicts.

replace $A \rightarrow aB \mid a$

with

$A \rightarrow a\ A'$

$A' \rightarrow B \mid \epsilon$

Left-Factoring a Grammar

- General procedure to left-factor a grammar:
  - For each non-terminal A, find the longest prefix α common to two or more of its alternatives.
    - So, α is a conflicting prefix
  - Replace all the A productions

    $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$ (where γ does not begin with α)

    by

    $A \rightarrow \alpha A' \mid \gamma$

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

Top-Down Parsing

- What about ε productions?
  - Consider $A \rightarrow \alpha$ and $A \rightarrow \beta$ and α may be empty
  - In this case there is no symbol to identify rhs α

Example:

- FIRST(rhs 3) = { ε }
- What lookahead symbol tells us we are matching rhs of 3?

#	*Production rules*			
0	*S*	→	A	B
1	*A*	→	x	B
2		|	y	C
3		|	*ε*	
4	*B*	→	z	

- Solution
  - Build a ***FOLLOW*** set for each production that can produce ε

Follow Sets

- **Follow(N)**, where N is a *non-terminal* symbol, is the *set of terminal symbols* that can follow immediately after any sentence that can be derived from any rule of N

- In this grammar:
  - E -> 0 | E + E
  - Follow(E) = {+, <EOF>}

Computation of Follow(T)

Examine all cases where the non-terminal T appears on the rhs of a rule in the grammar.

Follow(T) is the *smallest* set that satisfies the following rules.

- $N \rightarrow \alpha\, T$ or $N \rightarrow \alpha\, [\, T\,]$ or $N \rightarrow \alpha\, \{\, T\, \}$
  - Follow(N) $\subseteq$Follow(T)
- $N \rightarrow \alpha\, T\, \beta$ or $N \rightarrow \alpha\, [T]\, \beta$ or $N \rightarrow \alpha\, \{T\}\, \beta$
  - If $\epsilon \notin$ First(β) then First(β) $\subseteq$Follow(T)
  - If $\epsilon \in$ First(β) then ((First(β)$-\{\epsilon\}$) $\cup$ Follow(N)) $\subseteq$Follow(T)
- The Follow set of the start symbol contains <EOF> (or **$**).

Example: Calculating Follow Sets (1)

#	Production rule
1	*goal* → *expr*
2	*expr* → *term expr2*
3	*expr2* → + *term expr2*
4	| – *term expr2*
5	| ε
6	*term* → *factor term2*
7	*term2* → * *factor term2*
8	| / *factor term2*
9	| ε
10	*factor* → number
11	| identifier

FOLLOW(*goal*) = { $ }

FOLLOW(*expr*) = FOLLOW(*goal*) = { $ }

FOLLOW(*expr2*) = FOLLOW(*expr*) = {$}

FOLLOW(*term*) += (FIRST(*expr2*)\ {ε})∪ FOLLOW(*expr2)*

+= { +, – } ∪ FOLLOW(*expr2*)

+= { +, –, $ }

FOLLOW(*term2*) += FOLLOW(*term*)

FOLLOW(*factor*) += (FIRST(*term2*)\ {ε})∪ FOLLOW(term2*)*

+= { *, / } ∪ FOLLOW(*term2*)

+= { *, / , +, –, $ }

LL Grammar Restriction 3

- **Grammar Restriction 3:**

If a nonterminal may occur zero times (i.e. is optional), its First and Follow sets must be different (so we know whether to parse it or skip it on seeing a terminal symbol/token).

Summary recursive descent parsing

- Massage grammar to satisfy LL conditions
  - Remove left recursion
  - Left factor, whenever possible

- Build First (and Follow) sets

- Define a procedure for each non-terminal
  - Implement a case for each right-hand side
  - Recursively call procedures for non-terminals

Exercises

- Write (by hand) recursive descent parsers for the following three grammars
  - $S \rightarrow + S\ S \mid \text{-}\ \mathrm{S}\ \mathrm{S} \mid \mathrm{a}$
  - $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$
  - $S \rightarrow 0\ S\ 1 \mid 0\ 1$

- Download and install **`LLnextgen`**
  - http://os.ghalkes.nl/LLnextgen/index.html
  - Read its documentation if necessary.
  - Llnextgen is available on the lab computers.

- Remake the above exercises using **`LLnextgen`**.