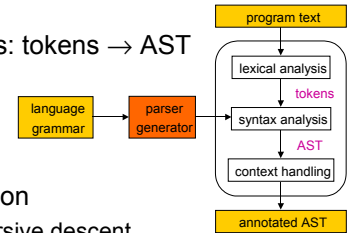


Delft University of Technology
The Netherlands

- AST construction
 - by hand: recursive descent
 - automatic: top-down (**LLgen**), bottom-up (**yacc**)



- $G = (V_N, V_T, S, P)$
 - V_N : set of Non-terminal symbols
 - V_T : set of Terminal symbols
 - S : Start symbol ($S \in V_N$)
 - P : set of Production rules $\{N \rightarrow \alpha\}$
- $V_N \cap V_T = \emptyset$
- $P = \{N \rightarrow \alpha \mid N \in V_N \wedge \alpha \in (V_N \cup V_T)^*\}$

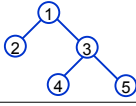
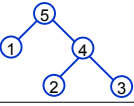
```

graph TD
    input --> expression
    input --> EOF
    expression --> term
    expression --> rest_expression
    term --> IDENTIFIER
    IDENTIFIER --> aap
  
```

```

graph TD
    rest_expression1[rest_expression] --> expression1[expression]
    rest_expression1 --> rest_expression2[rest_expression]
    expression1 --> term1[term]
    expression1 --> rest_expr1[rest_expr]
    term1 --> IDENT1[IDENT]
    IDENT1 --> aap[aap]
    rest_expr1 --> IDENT2[IDENT]
    IDENT2 --> noot[noot]
    rest_expr1 --> plus1[+]
    rest_expr1 --> rest_expr2[rest_expr]
    rest_expr2 --> IDENT3[IDENT]
    IDENT3 --> mies[mies]
    rest_expr2 --> epsilon[ε]
    rest_expr2 --> close_paren[)]
    rest_expression2 --> IDENT4[IDENT]
    IDENT4 --> aap2[aap]
    rest_expression2 --> plus2[+]
  
```

Comparison

	top-down	bottom-up
node creation	pre-order 	post-order 
alternative selection	first token	last token
grammar type	restricted LL(1)	relaxed LR(1)
implementation	manual + automatic	automatic

Recursive descent parsing

- each rule N translates to a boolean function
 - return **true** if a terminal production of N was matched
 - return **false** otherwise (without consuming any token)
- try alternatives of N in turn
- a terminal symbol must match the current token
- a non-terminal is matched by calling its routine

input \rightarrow expression EOF

```
int input(void) {
    return expression() && require(token(EOF));
}
```

Recursive descent parsing

expression \rightarrow term rest_expression

```
int expression(void) {
    return term() && require(rest_expression());
}
```

term \rightarrow IDENTIFIER | '(' expression ')'

```
int term(void) {
    return token(IDENTIFIER) ||
        token('(') && require(expression()) && require(token(')'));
}
```

Recursive descent parsing

rest_expression \rightarrow '+' expression | ϵ

```
int rest_expression(void) {
    return token('+') && require(expression()) || 1;
}
```

auxiliary functions

- consume matched tokens
- report syntax errors

```
int token(int tk) {
    if (Token.class != tk) return 0;
    get_next_token(); return 1;
}
```

```
int require(int found) {
    if (!found) error();
    return 1;
}
```

Automatic top-down parsing

- follow recursive descent scheme, but avoid interpretation overhead
- for each rule and alternative determine the tokens it can start with: **FIRST** set
- parsing scheme for rule $N \rightarrow A_1 | A_2 | \dots$
 - if token \notin FIRST(N) then ERROR
 - if token \in FIRST(A_1) then parse A_1
 - if token \in FIRST(A_2) then parse A_2
 - ...

Exercise (7 min.)

- design an algorithm to compute the FIRST sets of all non-terminals in a context free grammar.
- hint: consider the types of rules
 - alternatives
 - composition
 - empty productions

```
input  $\rightarrow$  expression EOF
expression  $\rightarrow$  term rest_expression
term  $\rightarrow$  IDENTIFIER | '(' expression ')'
rest_expression  $\rightarrow$  '+' expression |  $\epsilon$ 
```

Answers

Predictive parsing

- similar to recursive descent, but no back-tracking
- functions "know" what they are doing

input \rightarrow expression EOF $\text{FIRST}(\text{expression}) = \{\text{IDENT}, '('\}$

```
void input(void) {
    switch (Token.class) {
        case IDENT: case '(':
            expression(); token(EOF); break;
        default:
            error();
    }
}

void token(int tk) {
    if (Token.class != tk) error();
    get_next_token();
}
```

Predictive parsing

expression \rightarrow term rest_expression $\text{FIRST}(\text{term}) = \{\text{IDENT}, '('\}$

```
void expression(void) {
    switch (Token.class) {
        case IDENT: case '(':
            term(); rest_expression(); break;
        default:
            error();
    }
}
```

term \rightarrow IDENTIFIER | '(' expression ')'

```
void term(void) {
    switch (Token.class) {
        case IDENT: token(IDENT); break;
        case '(': token('('); expression(); token(')'); break;
        default: error();
    }
}
```

Predictive parsing

rest_expression \rightarrow '+' expression | ϵ $\text{FIRST}(\text{rest_expr}) = \{+, \epsilon\}$

```
void rest_expression(void) {
    switch (Token.class) {
        case '+': token('+'); expression(); break;
        case EOF: case ')': break;
        default: error();
    }
}
```

$\text{FOLLOW}(\text{rest_expr}) = \{\text{EOF}, '\}'$

- $\text{FIRST}(\epsilon) = \{\epsilon\}$
 - check nothing?
 - NO: token $\in \text{FOLLOW}(\text{rest_expr})$

Limitations of LL(1) parsers

- FIRST/FIRST conflict

term \rightarrow IDENTIFIER
 | IDENTIFIER '[' expression ']'
 | '(' expression ')'

- FIRST/FOLLOW conflict

$S \rightarrow A 'a' 'b'$

$A \rightarrow 'a' | \epsilon$ $\text{FIRST}(A) = \{ 'a' \} = \text{FOLLOW}(A)$

- left recursion

expression \rightarrow expression '.' term | term

Making grammars LL(1)

- manual labour
 - rewrite grammar
 - adjust semantic actions
- three rewrite methods
 - left factoring
 - substitution
 - left-recursion removal

Left factoring

term \rightarrow IDENTIFIER
 \mid IDENTIFIER '[' expression ']'

- factor out common prefix

term \rightarrow IDENTIFIER after_identifier
 after_identifier $\rightarrow \epsilon \mid$ '[' expression ']'

'[' \notin FOLLOW(after_identifier)

Substitution

$S \rightarrow A 'a' 'b'$
 $A \rightarrow 'a' \mid \epsilon$

- replace non-terminal by its alternative

$S \rightarrow 'a' 'a' 'b' \mid 'a' 'b'$

Left-recursion removal

$N \rightarrow N \alpha \mid \beta$

- replace by

$N \rightarrow \beta M$
 $M \rightarrow \alpha M \mid \epsilon$

- example

expression \rightarrow expression '-' term \mid term

expression \rightarrow term tail
 tail \rightarrow '-' term tail $\mid \epsilon$

β
 $\beta \alpha$
 $\beta \alpha \alpha$
 $\beta \alpha \alpha \alpha$
 ...

Exercise (7 min.)

- make the following grammar LL(1)

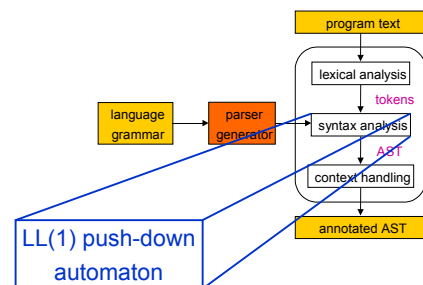
expression \rightarrow expression '+' term \mid expression '-' term \mid term
 term \rightarrow term '*' factor \mid term '/' factor \mid factor
 factor \rightarrow '(' expression ')' \mid func-call \mid identifier \mid constant
 func-call \rightarrow identifier '(' expr-list? ')' \mid identifier '('
 expr-list \rightarrow expression '(' ',' expression)*

- and what about

$S \rightarrow$ if E then S (else S)?

Answers

automatic generation



LL(1) push-down automaton

transition table

state (top of stack)	look-ahead token				
	IDENT	+	()	EOF
input	expression EOF		expression EOF		
expression	term rest-expr		term rest-expr		
term	IDENT		(expression)		
rest-expr		+ expression		ε	ε

- stack right-hand side of production

LL(1) push-down automaton

prediction stack input

input aap + (noot + mies) EOF

state (top of stack)	look-ahead token				
	IDENT	+	()	EOF
input	expression EOF		expression EOF		
expression	term rest-expr		term rest-expr		
term	IDENT		(expression)		
rest-expr		+ expression		ε	ε

LL(1) push-down automaton

prediction stack input

input aap + (noot + mies) EOF

replace non-terminal by transition entry

state (top of stack)	look-ahead token				
	IDENT	+	()	EOF
input	expression EOF		expression EOF		
expression	term rest-expr		term rest-expr		
term	IDENT		(expression)		
rest-expr		+ expression		ε	ε

LL(1) push-down automaton

prediction stack expression EOF

input aap + (noot + mies) EOF

replace non-terminal by transition entry

state (top of stack)	look-ahead token				
	IDENT	+	()	EOF
input	expression EOF		expression EOF		
expression	term rest-expr		term rest-expr		
term	IDENT		(expression)		
rest-expr		+ expression		ε	ε

LL(1) push-down automaton

prediction stack term rest-expr EOF

input aap + (noot + mies) EOF

replace non-terminal by transition entry

state (top of stack)	look-ahead token				
	IDENT	+	()	EOF
input	expression EOF		expression EOF		
expression	term rest-expr		term rest-expr		
term	IDENT		(expression)		
rest-expr		+ expression		ε	ε

LL(1) push-down automaton

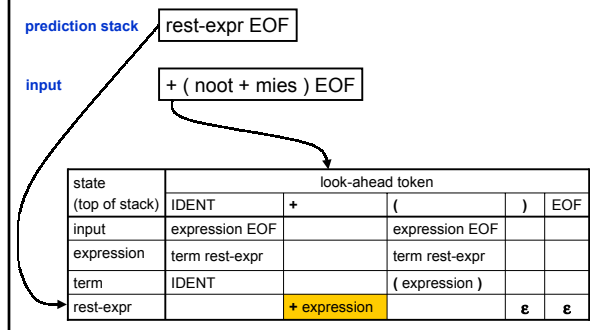
prediction stack IDENT rest-expr EOF

input aap + (noot + mies) EOF

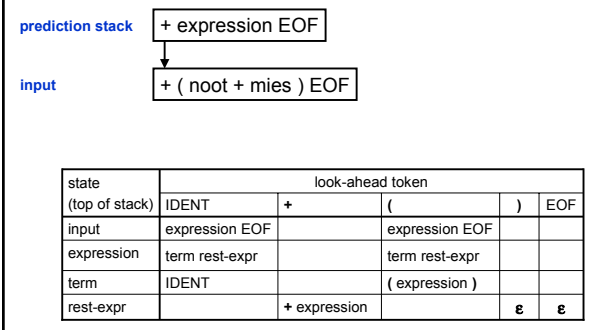
pop matching token

state (top of stack)	look-ahead token				
	IDENT	+	()	EOF
input	expression EOF		expression EOF		
expression	term rest-expr		term rest-expr		
term	IDENT		(expression)		
rest-expr		+ expression		ε	ε

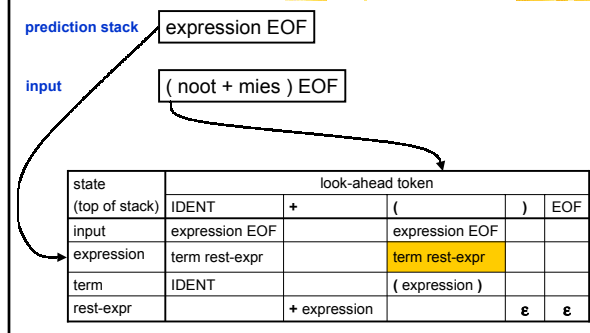
LL(1) push-down automaton



LL(1) push-down automaton



LL(1) push-down automaton

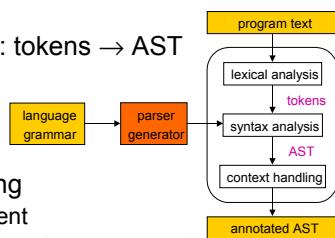


LLgen

- top-down parser generator
- to be used in assignment #1
- discussed in lecture 5

Summary

- syntax analysis: tokens → AST



- top-down parsing
 - recursive descent
 - push-down automaton
 - making grammars LL(1)

Homework

- study sections:
 - 1.10 closure algorithm
 - 2.2.4.6 error handling in LL(1) parsers
- print handout for next week [blackboard]
- find a partner for the “practicum”
- register your group
 - send e-mail to koen@pds.twi.tudelft.nl