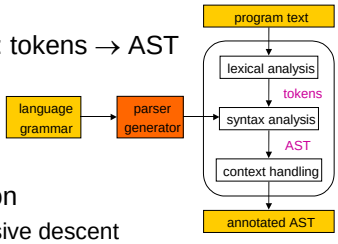
Compiler construction in4303 – lecture 3

Top-down parsing

Chapter 2.2 - 2.2.4

Overview

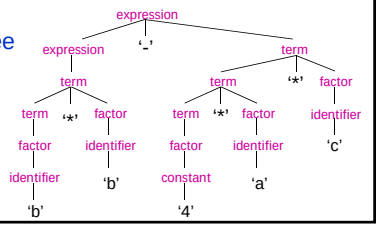
- syntax analysis: tokens → AST



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

Syntax analysis

- parsing: given a context-free grammar and a stream of tokens, find the derivation that ties them together.
- result: parse tree



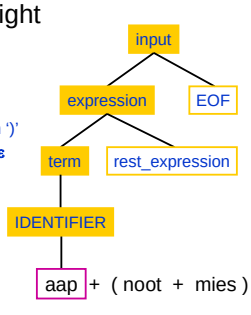
Context free grammar

- G = (V_N, V_T, S, P)
  - V_N : set of Non-terminal symbols
  - V_T : set of Terminal symbols
  - S : Start symbol (S ∈ V_N)
  - P : set of Production rules {N → α}
- $V_N \cap V_T = \varnothing$
- P = {N → α | N ∈ V_N ∧ α ∈ ($V_N \cup V_T$)*}

Top-down parsing

- process tokens left to right
- expression grammar

  input → expression EOF
  expression → term rest_expression
  term → IDENTIFIER | '(' expression ')'
  rest_expression → '+' expression | ε



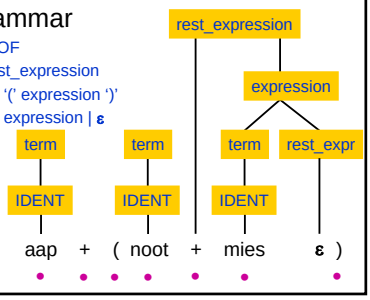
- example expression

aap + (noot + mies)

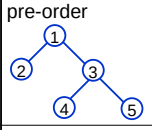
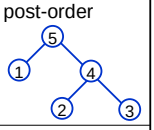
Bottom-up parsing

- process tokens left to right
- expression grammar

  input → expression EOF
  expression → term rest_expression
  term → IDENTIFIER | '(' expression ')'
  rest_expression → '+' expression | ε



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 → expression EOF

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

Recursive descent parsing

expression → term rest_expression

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

term → IDENTIFIER | '(' expression ')'

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

Recursive descent parsing

rest_expression → '+' 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 $N \rightarrow A_1 \mid A_2 \mid \ldots$
  - **if** token ∈ FIRST(A_1) **then** parse A_1
  - **else if** token ∈ FIRST(A_2) **then** parse A_2
  - …
  - **else** ERROR

Exercise (4 min.)

- determine the FIRST set of the non-terminals in our expression grammar

input → expression EOF
expression → term rest_expression
term → IDENTIFIER | '(' expression ')'
rest_expression → '+' expression | ε

- and, for this grammar

S → A B
A → A 'a' | ε
B → B 'b' | 'b'

Answers

Computing the FIRST set

- $N \rightarrow w \quad (w \in V_T)$
- $N \rightarrow A_1 \mid A_2 \mid \ldots$
- $N \rightarrow \varepsilon$
- $N \rightarrow A\ \beta$

closure algorithm:
- initial values
- inference rules
- iterative solution

Predictive parsing

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

input → expression EOF FIRST(expression) = {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 → term rest_expression FIRST(term) = {IDENT, '('}

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

term → IDENTIFIER | '(' expression ')'

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

Predictive parsing

rest_expression → '+' expression | **ε** FIRST(rest_expr) = {'+', **ε**}

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

FOLLOW(rest_expr) = {EOF, ')'}

- FIRST(ε) = {ε}
  - check nothing?
  - NO: token ∈ FOLLOW(rest_expr)

Limitations of LL(1) parsers

- FIRST/FIRST conflict

  term → IDENTIFIER
  | IDENTIFIER '[' expression ']'
  | '(' expression ')'

- FIRST/FOLLOW conflict

  S → A 'a' 'b'
  A → 'a' | ε

  FIRST(A) = { 'a', **ε** } ⊃ {'a'} = FOLLOW(A)

- left recursion

  expression → 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 → IDENTIFIER
| IDENTIFIER '[' expression ']'

- factor out common prefix

term → IDENTIFIER after_identifier
after_identifier → ε | '[' expression ']'

'[' ∉ FOLLOW(after_identifier)

Substitution

S → A 'a' 'b'
A → 'a' | ε

- replace non-terminal by its alternative

S → 'a' 'a' 'b' | 'a' 'b'

Left-recursion removal

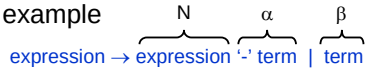
N → N α | β

- replace by

N → β M
M → α M | ε

β
β α
β α α
β α α α
...

- example



expression → term tail
tail → '-' term tail | ε

Exercise (7 min.)

- make the following grammar LL(1)

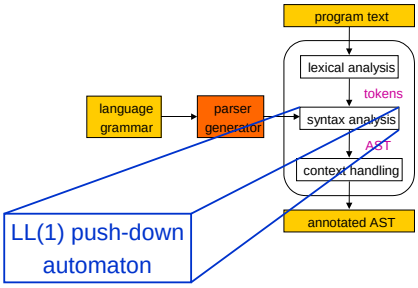
expression → expression '+' term | expression '-' term | term
term → term '*' factor | term '/' factor | factor
factor → '(' expression ')' | func-call | identifier | constant
func-call → identifier '(' expr-list? ')'
expr-list → expression (',' expression)*

- and what about

S → 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

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

pop matching token

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: rest-expr EOF

input: + (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: + (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