

Praktikum Compilerbau

Sitzung 3 – Parser

Prof. Dr.-Ing. Gregor Snelting
Andreas Zwinkau

IPD Snelting, Lehrstuhl für Programmierparadigmen



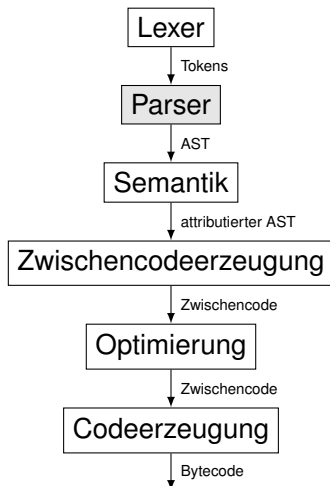
Wette gewonnen ;-)

- aber Software Engineering Mängel:
- Exit code $\neq 0$ bei Fehler
- Fehler: Invalid argument „-foo“
- Fehler: Permission denied

Testfall	Euer Compiler	C	D
manyidents	0,9s	0,3s	0,5s
speed0 (1Mb)	0,2s	0,0s	0,0s
speed1 (5Mb)	9,9s	0,7s	1,2s
speed2 (27Mb)	21,0s	4,2s	12,4s
speed3 (68Mb)	36min+	2,3s	7,0s

Problem: String-Concatenation

```
identifier += Character.toString((char) current);
```



Allgemein:

- Lesen des Tokenstroms von Links nach Rechts
- Finden der parserdefinierten Fehlerstelle

Praktikum:

- Von Hand implementierbar

Was ist SLL(k)?

Für $k \geq 1$ heißt eine kfG $G = (T, N, P, Z)$ eine SLL(k)-Grammatik (starke LL-Grammatik), wenn für alle Paare von Ableitungen

$$\begin{array}{ll}
 Z \Rightarrow_L^* \mu A \chi \Rightarrow \mu v \chi \Rightarrow^* \mu \gamma & \mu, \gamma \in T^*; v, \chi \in V^*, A \in N \\
 Z \Rightarrow_L^* \mu' A \chi' \Rightarrow \mu' \omega \chi' \Rightarrow^* \mu' \gamma' & \mu', \gamma' \in T^*; \omega, \chi' \in V^*
 \end{array}$$

gilt:

$$(k : \gamma = k : \gamma') \Rightarrow v = \omega$$

Also: Aus den nächsten k Zeichen kann **ohne** Berücksichtigung des Kellerinhalts die nächste anzuwendende Produktion eindeutig vorhergesagt werden.

SLL(k)-Grammatiken lassen sich mit rekursivem Abstieg implementieren. Beispiel für SLL(1):

1. Definiere Prozedur X für alle Nichtterminale X
2. Für alternative Produktionen $X \rightarrow X_1 \mid \dots \mid X_n$ sei Rumpf von X

```
switch t {  
  case Anf1(X1Folge1(X)) : Code für X1;  
  ...  
  case Anf1(XnFolge1(X)) : Code für Xn;  
  default : Fehler(...);  
}
```

3. ...

Rekursiver Abstieg 2

SLL(k)-Grammatiken lassen sich mit rekursivem Abstieg implementieren. Beispiel für SLL(1):

2. ...

3. Für rechte Seite $X_j = Y_1 \dots Y_m$ erzeuge:

$C_1; \dots; C_m$; **return**;

Es gilt $C_j =$

3.1 **if** ($t == Y_j$) $t = \text{nächstesSymbol}()$ **else** Fehler(...);

wenn $Y_j \in T$

3.2 $Y_j()$;

wenn $Y_j \in N$

Präzedenz und Links- bzw. Rechtsassoziativität kann über die Grammatik ausgedrückt werden.

$\text{Expr} ::= \text{AddSubExpr}.$

$\text{AddSubExpr} ::= (\text{AddSubExpr} ('+'|-'))? \text{MulDivExpr}.$

$\text{MulDivExpr} ::= (\text{MulDivExpr} ('*'|'/'))? \text{AtomicExpr}.$

$\text{AtomicExpr} ::= \text{Identifizier} \mid \text{Literal}.$

Beseitigen von Linksrekursion

$\text{Expr} ::= \text{AddSubExpr}.$

$\text{AddSubExpr} ::= (\text{AddSubExpr} ('+' | '-'))? \text{MulDivExpr}.$

$\text{MulDivExpr} ::= (\text{MulDivExpr} ('*' | '/'))? \text{AtomicExpr}.$

$\text{AtomicExpr} ::= \text{Identifier} \mid \text{Literal}.$

Variante 1

$\text{AddSubExpr} ::= \text{MulDivExpr} \text{AddSubExpr}' .$

$\text{AddSubExpr}' ::= (('+' | '-') \text{MulDivExpr} \text{AddSubExpr}')? .$

$\text{MulDivExpr} ::= \text{AtomicExpr} \text{MulDivExpr}' .$

$\text{MulDivExpr}' ::= (('*' | '/') \text{AtomicExpr} \text{MulDivExpr}')? .$

Variante 2

$\text{AddSubExpr} ::= \text{MulDivExpr} (('+' | '-') \text{MulDivExpr})^* .$

$\text{MulDivExpr} ::= \text{AtomicExpr} (('*' | '/') \text{AtomicExpr})^* .$

Für MiniJava ist die Liste länger:

```
Expression ::= AssignmentExpression .
AssignmentExpression ::= LogicalOrExpression ('=' AssignmentExpression)? .
LogicalOrExpression ::= (LogicalOrExpression '|')? LogicalAndExpression .
LogicalAndExpression ::= (LogicalAndExpression '&&')? EqualityExpression .
EqualityExpression ::= (EqualityExpression ('==' | '!='))? RelationalExpression .
RelationalExpression ::= (RelationalExpression ('<' | '<=' | '>' | '>='))? AdditiveExpression .
AdditiveExpression ::= (AdditiveExpression ('+' | '-'))? MultiplicativeExpression .
MultiplicativeExpression ::= (MultiplicativeExpression ('*' | '/' | '%'))? UnaryExpression .
UnaryExpression ::= PostfixExpression | ('!' | '-') UnaryExpression .
PostfixExpression ::= PrimaryExpression ( PostfixOp )* .
PostfixOp ::= MethodInvocation | FieldAccess | ArrayAccess .
MethodInvocation ::= '.' IDENT '(' Arguments ')' .
FieldAccess ::= '.' IDENT .
ArrayAccess ::= '[' Expression ']'.
Arguments ::= ( Expression (',' Expression)* )? .
PrimaryExpression ::= 'null' | 'false' | 'true' | INTEGER_LITERAL
    | IDENT | IDENT '(' Arguments ')' | 'this' | '(' Expression ')'
    | NewObjectExpression | NewArrayExpression .
NewObjectExpression ::= 'new' Type '(' ')' .
NewArrayExpression ::= 'new' Type '[' Expression ']' ( '[' ']' )* .
```

- Beim Parsen müssen viele Produktionen angewendet werden bis Identifier und Literale erkannt werden:

Expression → *AssignmentExpression* → ... → **IDENT**

- Das bedeutet für Parser mit rekursivem Abstieg viele Funktionsaufrufe.

Precedence Climbing

- elegante Lösung
- für jede Tokenklasse
 - Funktionszeiger für Postfix-/Infix-Parser
 - Funktionszeiger für Präfix-Parser
 - Präzedenz

```
typedef struct expression_parser_function_t {  
    parse_expression_function parser;  
    unsigned infix_precedence;  
    parse_expression_infix_function infix_parser;  
} expression_parser_function_t;  
expression_parser_function_t expression_parsers[T_LAST_TOKEN];
```

Precedence Climbing: Algorithmus

```
static expression_t *parse_subexpression(precedence_t precedence) {  
    expression_parser_function_t *parser = &expression_parsers[token.type];  
  
    /* parse prefix expression or primary expression */  
    expression_t *left;  
    if (parser->parser != NULL) left = parser->parser();  
    else left = parse_primary_expression();  
  
    while (true) {  
        parser = &expression_parsers[token.type];  
        if (parser->infix_parser == NULL || parser->infix_precedence < precedence)  
            break;  
  
        left = parser->infix_parser(left);  
    }  
    return left;  
}
```

Precedence Climbing: Linksassoziativ Infix

```
static expression_t *parse_add(expression_t *left)
{
    add_expression_t *result = allocate_add();
    result->left = left;
    next_token(); /* skip '+' */
    result->right = parse_subexpression(PRECEDENCE_ADD + 1);

    return result;
}
```

Precedence Climbing: Rechtsassoziativ Präfix

```
static expression_t *parse_prefix_plus_plus(void)  
{  
    prefix_plus_plus_t *result = allocate_prefix_plus_plus();  
  
    next_token(); /* skip '++' */  
    result->value = parse_subexpression(PRECEDENCE_PREFIX_PLUS_PLUS);  
  
    return result;  
}
```


Vorteile von Precedence Climbing

- Effizient (-er als naives LL)
- Einfache Spezifikation der Ausdrücke
- Einfacher Algorithmus
- Operatoren lassen sich dynamisch anlegen (z.B. für Haskell nötig)