Karlsruher Institut für Technologie

Praktikum Compilerbau
Sitzung 3 – Parser

Prof. Dr.-Ing. Gregor Snelting
Matthias Braun und Sebastian Buchwald

IPD Snelting, Lehrstuhl für Programmierparadigmen

Inhalt

Karlsruher Institut für Technologie

1. Altes Übungsblatt
2. Einordnung des Parsers
3. Theorie
4. Praxis
    - Rekursiver Abstieg
    - Precedence Climbing

Altes Übungsblatt

Karlsruher Institut für Technologie

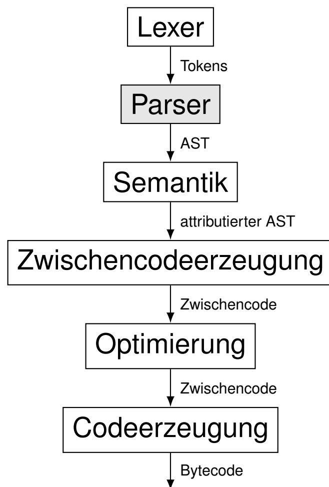
Wette gewonnen ;-)

- Fehlermeldung für leere Datei
- Endlosschleife bei Fehlern
- ’0’ ist eigenes Token
- Fehlende Operatoren (`+=`, `-=`, `|`)
- `int/**/i;`
- `-2147483648`
- `/*`
- Abbruch bei `EOF`
- Ausgabeformat: kein `error:`, `=*`, ˆ(seltsames unicode =)

Performance

Testfall	Compiler 1	Compiler 2	Compiler 3	naiv C++	C
manyidents	0,9s	1,8s	0,9s	0,3s	0,0s
speed0 (1Mb)	0,2s	S.-Overflow	0,1s	0,0s	0,0s
speed1 (5Mb)	3,5s	(10,3s)	3,0s	0,7s	0,1s
speed2 (27Mb)	18,4s	(63,2s)	16,7s	4,0s	0,8s
speed3 (68Mb)	4,2s	>300s	4,2s	2,1s	0,5s

Phasen



Anforderungen

Karlsruher Institut für Technologie

Allgemein:

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

Praktikum:

- Von Hand implementierbar

Was ist $\mathrm{SLL}(k)$?

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

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

gilt:

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

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

Rekursiver Abstieg 1

Karlsruher Institut für Technologie

$\mathrm{SLL}(k)$-Grammatiken lassen sich mit rekursivem Abstieg implementieren. Beispiel für $\mathrm{SLL}(1)$:

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

   switch t {
     case $\mathrm{Anf}_1(X_1 \mathrm{Folge}_1(X))$: Code für X_1;
     ...
     case $\mathrm{Anf}_1(X_n \mathrm{Folge}_1(X))$: Code für X_n;
     default : Fehler(...);
   }

3. ...

Rekursiver Abstieg 2

Karlsruher Institut für Technologie

$\mathrm{SLL}(k)$-Grammatiken lassen sich mit rekursivem Abstieg implementieren. Beispiel für $\mathrm{SLL}(1)$:

2. ...
3. Für rechte Seite $X_i = Y_1 \dots Y_m$ erzeuge:

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

   Es gilt $C_i =$

   3.1 **if** (t == Y_i) t = nächstesSymbol() **else** Fehler(...);

   wenn $Y_i \in T$

   3.2 $Y_i()$;

   wenn $Y_i \in N$

Ausdrücke

Karlsruher Institut für Technologie

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

Expr ::= AddSubExpr.
AddSubExpr ::= (AddSubExpr ('+'|'−'))? MulDivExpr.
MulDivExpr ::= (MulDivExpr ('∗'|'/'))? AtomicExpr.
AtomicExpr ::= Identifier | Literal.

Beseitigen von Linksrekursion

Karlsruher Institut für Technologie

Expr ::= AddSubExpr.
AddSubExpr ::= (AddSubExpr ('+'|'−'))? MulDivExpr.
MulDivExpr ::= (MulDivExpr ('∗'|'/'))? AtomicExpr.
AtomicExpr ::= Identifier | Literal.

Variante 1

AddSubExpr ::= MulDivExpr AddSubExpr' .
AddSubExpr' ::= (('+'|'−') MulDivExpr AddSubExpr')?.
MulDivExpr ::= AtomicExpr MulDivExpr'.
MulDivExpr' ::= (('∗'|'/') AtomicExpr MulDivExpr')?.

Variante 2

AddSubExpr ::= MulDivExpr (('+'|'−') MulDivExpr)∗.
MulDivExpr ::= AtomicExpr (('∗'|'/') AtomicExpr)∗.

Mehr Operatoren

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 ']' ( '[' ']' )∗ .
```

Optimierungspotential

Karlsruher Institut für Technologie

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

$$\textit{Expression} \rightarrow \textit{AssignmentExpression} \rightarrow \ldots \rightarrow \texttt{IDENT}$$

- Das bedeutet für Parser mit rekursivem Abstieg viele Funktionsaufrufe.

Precedence Climbing

Karlsruher Institut für Technologie

- 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

Karlsruher Institut für Technologie

```
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

KIT
Karlsruher Institut für Technologie

```
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 Prefix

Karlsruher Institut für Technologie

```
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

Karlsruher Institut für Technologie

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