Compilerbau: Projekt Turtle Graphics - Hinweise zu Lex & Yacc

Klaus Kusche

- Im Vergleich zu unserem Taschenrechner ist der Typ **YYSTYPE** hier komplexer:

  Er definiert ja den *Typ des Wertes*, der jedem *Token* und jedem *Nonterminal* zugeordnet ist. Hier treten auf den ersten Blick *mehrere verschiedene Typen* auf:

  - *Zahlkonstanten*-Token haben einen **`double`**-*Wert*
  - *Namens*-Token haben einen *Pointer in die Namenstabelle* als Wert
  - Alle *Nonterminals* haben einen *Pointer auf einen Syntaxbaum-Knoten* als Wert

  Das Problem kann man auf *zwei Wegen* lösen:

  - Entweder man nutzt eine **`union`** dieser drei Typen,
    wahlweise durch Definition eines eigenen **`union`**-Typs in C
    oder mittels des **`%union`**-Konstruktes von Lex und Yacc.
  - Oder man verwendet *einheitlich* einen *Syntaxbaumknoten-Pointer* als **YYSTYPE**
    und legt *gleich im Lexer* einen **`name_any`**-*Baumknoten* für allen *Namen*
    bzw. einen **`oper_const`**-Baumknoten für numerische *Konstanten* an.

    Für solche **`name_any`**-Knoten vom Lexer kann es passieren,
    dass man dann *im Parser* die Art des Knotens noch einmal *ändern* muss
    (z.B. für Funktions-Aufrufe, Zuweisungs-Befehle oder Zählschleifen),
    oder dass man den *Namenspointer* aus dem Knoten *entnimmt*
    und den Knoten dann gleich wieder *freigibt* (z.B. für Funktions-Parameter).
  - Eine Sonderbehandlung brauchen die Nonterminals *params* und *args*, weil sie ja eigentlich *keinen Baumknoten* liefern, sondern *in ein Array* eingetragen werden.
    Wenn Yacc *params* oder *args* parst, ist dieses Array aber *noch nicht bekannt*.

    Ich empfehle daher Folgendes:

    - Im Code der Regeln von *params* und *args* die *einzelnen Parameter* bzw.
      Argumente über das **`next`**-Member ihrer Knoten *zu einer verketteten Liste*
      zusammenhängen und den *ersten Knoten der Liste als Wert* von *params* und
      args zu speichern (der passt auch zum Wert-Typ **YYSTYPE**).
    - Im *Code der übergeordneten Regeln* diese Liste *durchlaufen*
      und die Werte in das entsprechende Array *übertragen*.
- Eindeutigkeit und Auswahl von Regeln in Lex:
  Bei der *Unterscheidung* zwischen *Namen und Schlüsselwörtern*
  (oder z.B. zwischen **<** und **<=**) in Lex hilft dir Folgendes:
  - Lex wählt an der aktuellen Position im Input immer diejenige Regel,
    deren Regular Expression *den **längstmöglichen** passenden Text* ergibt.
  - Gibt es *zwei* passende Regeln, die im Input ein *gleichlanges Textstück* erkennen,
    wird die gewählt, die *im Lex-File **weiter oben*** steht. Man sollte daher
    die einzelnen Schlüsselwörter *vor* der Regel für allgemeine Namen definieren,
    dann werden sie als Schlüsselwort und nicht als Name erkannt.

- Wie beim handgeschriebenen Compiler sollte Lex für jedes *Schlüsselwort* ein *eigenes Token* liefern. Für *alle Namen* (incl. aller vordefinierten Variablen und Funktionen) sollte Lex hingegen ein *einheitliches Namens-Token* liefern.
- **`yyparse`** liefert als Returnwert standardmäßig nur eine *Erfolgsanzeige*.
  Am besten speichert man das *eigentliche Ergebnis* des Parsers
  (= die Wurzel unseres Syntaxbaumes) im *Code der Regel des Startsymbols*
  (ganzen Programms) in einer selbst deklarierten *globalen Variablen*.

Fehlerbehandlung:

- Wir leben damit, dass Yacc mittels **`yyerror`** nur *„Syntax Error“* ausgibt,
  wenn der Input nicht den Syntaxregeln entspricht
  (wenn ihr *eigene Fehlermeldungen* generiert,
  solltet ihr aber einen *besseren* Text ausgeben!).
- Es sollte jedoch schon euer Ehrgeiz sein,
  in jeder Fehlermeldung *sinnvolle Zeilen- und Spaltennummer* auszugeben,
  sowohl im vordefinierten **`yyerror`** als auch in euren *eigenen Fehlermeldungen*.

  Außerdem sollte in jedem *Syntaxbaum-Knoten*
  das Member **`srcpos_t pos`** sinnvoll *befüllt* werden!

Für die *Quelltext-Position* enthalten Flex und Bison zwar *rudimentäre Logik*,
aber diese ist leider lückenhaft und passt nicht so recht zusammen:

- Flex zählt *auf Wunsch* die aktuelle *Zeilennummer* (aber *keine* Spaltennummer)
  in der globalen Variable **`yylineno`** mit.

  Dieses Feature muss man ganz oben im Flex-File *einschalten*:

  `%option yylineno`
- Bison definiert *auf Wunsch* einen Typ **YYLTYPE** für *Source-Positionen*:

  ```
  typedef struct YYLTYPE
  {
    int first_line;
    int first_column;
    int last_line;
    int last_column;
  } YYLTYPE;
  ```
  (es wäre zwar möglich, diesen Typ umzudefinieren, es bringt aber für uns wenig)

  Um den **YYLTYPE**-Mechanismus zu *aktivieren*,
  schreibt man oben im Yacc-File *nach* dem **`%{`** … **`}%`**-Block **`%locations`** .
- Bison verwaltet *für jedes Terminal oder Nonterminal dessen Position*,
  und zwar ausgehend von der *vordefinierten globalen Variable* **`yylloc`**:
  Von dieser *erwartet* Bison, dass sie die *Position des aktuellen Tokens* enthält.

  Dann erkennt Bison im Code jeder Yacc-Regel
  analog zu **`$1`**, **`$2`**, … für den Code des *i*-ten Konstruktes der rechten Regelseite
  auch **`@1`**, **`@2`**, … für die *Position des i-ten Konstruktes* der rechten Regelseite:
  Das **`@`*`i`*** wird durch den **YYLTYPE**-Wert des *i*-ten Konstruktes ersetzt
  (um diesen Wert z.B. effizient als Parameter zu übergeben, kann man auch
  `& @`*`i`* verwenden, um einen **YYLTYPE**-Pointer darauf zu bekommen).

Aus diesem **YYLTYPE-@*i***-Wert kann man die *Position* für schöne *Fehlermeldungen* oder für unsere **srcpos_t**-*Werte* entnehmen: **first_line** und **first_column**.

Damit man **YYLTYPE** und **yylloc** nutzen kann, muss *jeder betroffene C-File* den von Bison generierten Header ***xxx*.tab.h** *inkludieren*, insbesondere auch der *Lex-File* in seinem **%{** … **}%**-Block (für die **yylloc**-Berechnung)

- Für **yyerror** kann man **yylloc** direkt verwendet, z.B.

```
void yyerror(const char *s)
{
  fprintf(stderr, "%s at line %d, column %d\n", s,
        yylloc.first_line, yylloc.first_column);
}
```

- Leider enthalten weder Flex noch Bison Code, um **yylloc** richtig zu setzen:
  Flex kennt **yylloc** nicht, und Bison liest den Wert nur.
  Um **yylloc** für jedes Token zu berechnen, muss man selbst Code schreiben, und zwar *im Lex-File* innerhalb des **%{** … **}%**-Blocks:

```
#include <string.h>
#define YY_USER_ACTION                                          \
  yylloc.first_line = yylloc.last_line;                         \
  yylloc.first_column = yylloc.last_column;                     \
  if (yylloc.last_line == yylineno) {                           \
    yylloc.last_column += yyleng;                               \
  } else {                                                      \
    yylloc.last_line = yylineno;                                \
    yylloc.last_column = yytext + yyleng - strrchr(yytext, '\n'); \
  }
```

  (nach den *Zeilenfortsetzungs*-\ am Ende der Zeilen muss *gleich das Newline* stehen)

  Flex führt das Makro **YY_USER_ACTION** automatisch *bei jedem Token* aus.