Lemon Parsergenerator

Dorian Weber

Aufbau, Funktionsweise
und Vergleich mit Alternativen
für den Lemon Parsergenerator

Übersicht

- generelle Eigenschaften
- Installation
- Dateiaufbau
- Bedienung und Parameter
- Aufrufinterface
- Demonstration
- Diskussion

generelle Eigenschaften

- tabellengesteuert (LALR)
- optimiert für Parallelbetrieb / objektorientierter Aufbau
- Anti-Memory-Leak Strategie

generelle Eigenschaften

- generiert effizienten C-Code
- GPL'd; generierter Parser nicht unter GPL
- plattformunabhängig, durch Vergabe des C-Quellcodes und Verzicht auf Einbindung systemgebundener Bibliotheken

Installation

- zu http://www.hwaci.com/sw/lemon browsen
- lemon.c und lempar.c herunterladen
- unter Windows mit einem beliebigen Compiler, vorzugsweise MinGW compilieren

Installation

- unter Linux erst die Zeilenenden auf Unix (LF) umstellen und dann

```
gcc lemon.c -o lemon
```

- tempar.c in Verzeichnis der ausführbaren Datei legen

Dateiaufbau

- keine getrennten Sektionen, Deklarationen überall möglich
- C/Cpp-Style Kommentare erlaubt

Dateiaufbau

- Terminalsymbole (T) müssen nicht deklariert werden – markantes Merkmal: erster Buchstabe groß
- Nichtterminalsymbole (NT): kleiner Anfangsbuchstabe
- Syntax für Nichtterminale:

```
<name> ::= <part1> <part2>.
```

$$partN \in (T \cup NT)$$

Dateiaufbau

- Generatordirektiven ähnlich wie in Bison; Assoziativitäten identisch
  - `%left`

    ```
    a AND b AND c -> (a AND b) AND c
    ```
  - `%right`

    ```
    a EXP b EXP c -> a EXP (b EXP c)
    ```
  - `%nonassoc`

    ```
    a EQ b EQ c -> Fehler
    ```

Dateiaufbau

- Präzedenzen der Operatoren werden wie in Bison über die Reihenfolge der Assoziativitäten ausgedrückt
  - Präzedenzstufen steigen mit der Zeilennummer
  - Beispiel:

```
%left AND.
%left OR.
%nonassoc EQ NE.
%left ADD SUB.
%left MUL DIV.
%right EXP.
```

Dateiaufbau (Direktiven)

- Standarddirektiven
  - `%include` fügt C-Code an den Anfang des generierten Parsers
  - `%token_type` spezifiziert den Datentyp der Terminalsymbole (Standard: int)
  - `%type` analog für Nichtterminalsymbole

Dateiaufbau (Anti-Memory-Leak Strategie)

- Spezialdirektiven
  - `%token_destructor` führt Code aus, wenn ein Terminalsymbol reduziert wird
    dient der Freigabe von alloziertem Speicher
  - `%destructor` funktioniert analog für Nichtterminalsymbole

Dateiaufbau (Anti-Memory-Leak Strategie)

- `%parse_accept` führt genau dann Code aus, wenn der Parsevorgang erfolgreich war
- `%parse_failure` führt Code aus, wenn der Text nicht der Syntaxdefinition genügt
- `%stack_overflow` führt Code aus, wenn der interne Parserstack überläuft

Dateiaufbau

- Stackmanipulation prinzipiell wie in Bison
- Umsetzung geschickter – keine Durchnummerierung der Parameter für Klauseln

  Beispiel:

```
expr(A) ::= expr(B) PLUS expr(C)
{ A = B+C; }
expr(A) ::= expr(B) PLUS expr(C)
{ A = B; } <- Fehler
```

Bedienung und Parameter

- Aufruf wie gehabt `./lemon -<opts>`
- generiert drei Dateien
  - C-Code für den Parser
  - Headerdatei mit der Definition der Token für den Lexer
  - Reportdatei

Bedienung und Parameter

- nützliche Kommandozeilenparameter sind
  c : komprimiert nicht die Aktionstabelle, Fehler werden früher erkannt, Tabelle ist etwas größer
  q : unterdrückt die Reportdatei (d.h. nur Header und Source werden generiert)

Aufrufinterface

- zunächst Allozierung von Speicher für den Parser

```
void* pParser = ParseAlloc(malloc);
```

- Aufruf des Parsers mit Übergabe des nächsten Token, Wert des aktuellen Tokens und optionalem vierten Argument

```
Parse(pParser, iTokenID, pTokenData,
  pArg);
```

- zuletzt Freigabe des Parserspeichers

```
ParseFree(pParser, free);
```

Aufrufinterface (Beispiel)

```
ParseFile()
{
  pParser = ParseAlloc(malloc);

  while (NextToken(pTokenizer, &iTokenID, &pToken))
  {
    Parse(pParser, iTokenID, pToken);
  }

  Parse(pParser, 0, pToken);
  ParseFree(pParser, free);
}
```

Demonstration

- kurze Demo: „klassischer“ Taschenrechner

Vor- und Nachteile des Generators

- Vorteile:
  - erzeugt schnellen C-Quellcode
  - intuitive Bedienung
  - modular geschrieben
  - geeignet für Dauerbetrieb
  - übersichtlich und gut dokumentierter Quellcode

Vor- und Nachteile des Generators

- Nachteile:
  - vergleichsweise wenig Hilfe (im Vergleich mit Bison)
  - kein Lexer integriert und keine Lexer/Parser Kombination vorgesehen
  - wenig verfügbare Quellcodes im Vergleich mit Flex/Bison