`burg`, `iburg` und `bfe`

1 Einleitung

`burg` [FHP92] und `iburg` [FHP93] sind Programme die aus Baumgrammatiken Baumparser erzeugen. Die erzeugten Baumparser finden immer einen Parse mit minimalen Kosten und sind daher insbesondere für die Befehlsauswahl geeignet. Die von `iburg` erzeugten Parser verwenden dabei die im Vorlesungsskriptum beschriebene Methode, während die von `burg` erzeugten Parser als Baumautomaten implementiert und daher etwas schneller sind. Beide Baumparsergeneratoren akzeptieren die gleiche Eingabe und die erzeugten Parser finden gleich gute Ableitungen. Im Prinzip sind sie so entworfen, daß sie kompatipel miteinander sind. Leider ist diese Kompatibilität unter heutigen 64-bit Betriebsystemen nicht gegeben. Daher ist im Rest dieses Handbuchs praktisch ausschließlich von `iburg` die Rede, und Sie sollten auch `iburg` verwenden.

Der erzeugte Baumparser macht zwei Durchgänge durch den Baum: Im ersten Durchgang (Labeller) wird für jeden Knoten ein Zustand (bei `iburg` ähnlich dem Zustand eines endlichen Automaten) berechnet, anfangend mit den Blättern; im zweiten Durchgang (Reducer) wird ein optimaler Ableitungsbaum durchwandert, wobei gegebenenfalls entsprechende Aktionen durchgeführt werden können.

Die Eingabe von `iburg` ist auf maximale Flexibilität ausgelegt, und daher etwas unbequem. Sie ist eher dafür gedacht, als Zwischendarstellung für einen Codegeneratorgenerator zu dienen als direkt von Hand geschrieben zu werden. `bfe` ist ein front-end für `iburg`, das auf die in der Übung vorkommende Aufgabenstellung zugeschnitten ist und den Benutzern einige der Arbeiten abnimmt, die sie bei direkter Verwendung von `iburg` machen müßten.

```
%{
 enum { Assign=1, Constant, Fetch, Four, Mul, Plus };

 typedef struct tree {
         int op;
         struct tree *kids[2];
         STATEPTR_TYPE state;
 } *NODEPTR_TYPE;

 #define OP_LABEL(p)    ((p)->op)
 #define LEFT_CHILD(p)  ((p)->kids[0])
 #define RIGHT_CHILD(p) ((p)->kids[1])
 #define STATE_LABEL(p) ((p)->state)
 #define PANIC printf
 %}
 %start reg
 %term  Assign=1 Constant=2 Fetch=3 Four=4 Mul=5 Plus=6
 %%
 con:    Constant                = 1 (0);
 con:    Four                    = 2 (0);
 addr:   con                     = 3 (0);
 addr:   Plus(con,reg)           = 4 (0);
 addr:   Plus(con,Mul(Four,reg)) = 5 (0);
 reg:    Fetch(addr)             = 6 (1);
 reg:    Assign(addr,reg)        = 7 (1);
 %%
```

Abbildung 1: Eine Baumgrammatik

2 Der Aufbau der `iburg`-Eingabe

Burg erstellt aus einer Baumgrammatik als Eingabe einen Baum-Parser in C, der BURM genannt wird. Die Grammatik von `iburg` gleicht strukturell der von YACC. Kommentare werden wie in C geschrieben.

Das folgende Programmstück zeigt eine Beispielgrammatik, mit der eine sehr einfache Befehlsauswahl realisiert werden kann.

Abb. 2 zeigt eine EBNF-Gramatik für `iburg`-Eingaben, insbesondere für die Baumgrammatik. Kommentare, der Text zwischen "`%{`" und "`%}`" und der Text nach dem zweiten optionalen "`%%`" (User-Code) wird rein lexika-

```
grammar: {dcl} '%%' {rule} [ '%%' ]

dcl: '%start' Nonterminal
dcl: '%term' { Operator '=' ESN_Integer }

rule: Nonterminal ':' tree '=' ERN_Integer [ cost ] ';'

cost: /* empty */
cost: '(' Integer { ',' Integer } ')'

tree: Operator '(' tree ',' tree ')'
tree: Operator '(' tree ')'
tree: Operator
tree: Nonterminal
```

Abbildung 2: EBNF-Grammatik für `iburg`-Baumgrammatiken

lisch behandelt und daher in der EBNF-Grammatik nicht berücksichtigt. Nonterminale und Operatoren sind Identifier, wobei Operatoren vorher mit "`%term`" deklariert wurden. ERN steht für externe Regelnummer und ESN für externe Symbolnummer (s.u.).

Es folgt eine genauere Beschreibung der Einzelteile der `iburg`-Eingabe:

2.1 Die Configuration Section

Text zwischen "`%{`" und "`%}`" (sogenannte Configuration Sections) wird direkt in den erzeugten Baum-Parser kopiert. Da das der Scanner von `iburg` übernimmt, scheinen Configuration Sections nicht in Abb. 2 auf.

Üblicherweise enthalten Configuration Sections Deklarationen, Makros und/oder "`#include`"s, die der Baum-Parser braucht:

Der Typ `NODEPTR_TYPE` dekariert einen Zeiger auf einen Baumknoten. Der Baum-Parser greift auf den Operator eines Baumknoten `p` mit `OP_LABEL(p)` (ganzzahlig), und auf seine Kinder mit `LEFT_CHILD(p)` (`NODEPTR_TYPE`) und `RIGHT_CHILD(p)` (`NODEPTR_TYPE`) zu.

Der Baum-Parser berechnet und speichert einen Zustand in jedem Knoten des geparsten Baums. Er greift auf diesen Zustand über das Makro `STATE_LABEL(p)` (l-value vom Typ STATEPTR_TYPE) zu.

Bei Auftreten eines Fehlers wird `PANIC` als Routine im printf-Format aufgerufen.

2.2 Die Baumgrammatik

Nach der Configuration Section folgt die Baumgrammatik.

Eine Baum-Grammatik ähnelt einer kontextfreien Grammatik: sie besteht aus Regeln, Operatoren (Terminalen), Non-Terminalen und einem speziellen Start-Nonterminal. Im Unterschied zu normalen Grammatiken leiten Baumgrammatiken Bäume ab und nicht Strings (daher nennen wir die Terminale auch Operatoren). Die rechte Seite einer Regel, Baummuster genannt, ist ein Baum, der Nonterminale enthalten kann. Baummuster werden in geklammerter Prefix-Notation dargestellt. Regeln, deren Baummuster nur aus einem Nonterminal besteht, nennt man Kettenregeln.

2.2.1 Die Declaration Section

In der Declarations Section werden das Start-Nonterminal und die Operatoren (Terminale) der Bäume deklariert.

Alle Operatoren müssen deklariert werden, eine solche Deklarationszeile beginnt mit `%term`. Jeder Operator hat eine festgelgte Anzahl von Kindern (maximal zwei), die die Benutzer festlegen, indem sie den Operator entsprechend in Baummustern verwenden. Jedes Terminal hat eine eindeutigen externe Symbolnummer (ESN), einer positiven ganzen Zahl, definiert. `OP_LABEL(p)` muß diese externe Symbolnummer für den Knoten, auf den p zeigt, liefern.

Non-Terminale werden nicht deklariert, mit Ausnahme des Startsymboles, welches optional in einer Zeile, die mit "`%start`" beginnt, deklariert werden kann. Gibt es keine "`%start`"-Deklaration, verwendet `iburg` das Nonterminal der ersten Regel als Startsymbol.

2.2.2 Die Rules Section

Nach dem ersten "`%%`" beginnt die Rules Section, in der die Regeln der Baum-Grammatik definiert werden.

Eine Regel besteht aus einem Nonterminal auf der linken Seite, gefolgt von einem Doppelpunkt und einem Baummuster auf der rechten Seite, der externen Regelnummer (ERN) und einem Kostenvektor.

Eingebettete semantische Aktionen wie in YACC oder Attributierungen wie in Ox sind in `iburg` nicht erlaubt. Stattdessen wird jeder Regel eine sog. externe Regelnummer (ERN) zugeordnet. Im Reducer-Durchgang des Baumparsers können dann aufgrund dieser Regelnummer entsprechende Aktionen durchgeführt werden.

Am Ende einer Regel steht der Kostenvektor, ein Vektor von positiven, ganzzahligen Kosten, in Klammern und durch Kommas getrennt. Wird der Kostenvektor weggelassen, werden die Kosten als 0 angenommen. `iburg` berücksichtigt nur die ersten vier Elemente des Kostenvektors. Die Kosten einer Ableitung ergeben sich aus der Summe der Kosten der Regel selbst, und den Kosten aller Subregeln. Im Standardfall berücksichtigt `iburg` nur das erste Element des Kostenvektors, die sog. *Principal Costs*. Alternativen dazu können durch Aufrufoptionen von `iburg` ausgewählt werden.

Wenn kein Startsymbol deklariert wurde, verwendet `iburg` das Non-Terminal der ersten Regel als Startsymbol.

2.3 Anwendungscode

Der nach dem zweiten "`%%`" stehende Text wird direkt in den erzeugten Baumparser kopiert. Das macht wieder der Scanner von `iburg` und ist daher in Abb. 2 nicht zu sehen.

In diesem Abschnitt können z.B. Routinen wie der Reducer stehen.

3 Der Parser

Während im vorigen Kapitel mit der Baumgrammatik die statische Komponente eines `iburg`-Programmes beschrieben wurde, werden in diesem Kapitel die Funktionsweise und Realisierung des erzeugten Baumparsers erläutert.

Von den beiden Durchgängen des Labellers wird der erste (**Labeller**) auf Wunsch automatisch erzeugt. Die Benutzer müssen sich nur noch um den Aufruf kümmern.

Der zweite Durchlauf (**Reducer**) durchläuft den Ableitungsbaum, von der Wurzel ausgehend. Da das genaue Vorgehen hier von der Anwendung abhängt, stellt `iburg` nur einige Grundroutinen und Datenstrukturen zur Verfügung und überläßt dem Benutzer, einen Reducer zu schreiben. `bfe` (siehe Abschnitt 5) erzeugt Reducer, die für die Aufgabenstellungen in dieser Übung passen sollten.

Der von `iburg` generierte Code stellt Grundroutinen für Labeller und Reducer zur Verfügung, sodaß die Benutzer auf Wunsch z.B. selbst eine Labelling-Routine schreiben können. Außerdem generiert `iburg` einige Hilfsroutinen, die gebräuchliche Kombinationen der Grundroutinen beinhalten bzw. nützlich zum Debugging sind.

3.1 Der Labeller

Der von `iburg` generierte Labeller heißt `burm_label` und hat folgenden Typ:

```
extern STATEPTR_TYPE burm_label(NODEPTR_TYPE p);
```

Diese Routine führt das Labeln des gesamten Baumes, auf den `p` zeigt, durch, und liefert den Zustand (`STATE_LABEL(p)`) von `p` zurück. Bäume, die nicht abgeleitet werden können, erhalten den State 0.

Den Rest dieses Abschnitts brauchen sie nur lesen, wenn Sie selbst einen Labeller schreiben wollen.

Die Grundroutine `burm_state` beinhaltet zum Unterschied von `burm_label` keinen Code zum Durchlaufen des Baumes und zum Beschreiben der Felder. Sie kann zum Labeln verwendet werden, wenn der Programmierer eigenen Code zum Durchlaufen des Baumes ausführen will.

```
extern STATEPTR_TYPE burm_state(int op,
       STATEPTR_TYPE leftstate, STATEPTR_TYPE rightstate);
```

Die Parameter sind eine externe Symbolnummer für einen Knoten und die Labels für das linke und rechte Kind des Knotens. Sie liefert das State Label, das dem jeweiligen Knoten zuzuweisen ist. Bei unären Operatoren wird `rightstate` ignoriert, bei Blättern werden `left-` und `rightstate` ignoriert.

3.2 Der Reducer

Sie brauchen diesen Abschnitt nur zu lesen, wenn Sie selbst einen Reducer schreiben wollen, statt den von `bfe` erzeugten zu verwenden.

Die wichtigste Routine für den Reducer ist `burm_rule`. Sie hat folgende Form:

```
extern int burm_rule(STATEPTR_TYPE state, int goalnt);
```

Die Parameter sind `state`, der Zustand (`STATE_LABEL(p)`) des Baumes und `goalnt`, eine Zahl, die das Zielnonterminal identifiziert, aus dem der Baum abgeleitet werden soll. Die Zahl des Startnonterminals ist 1. `burm_rule` liefert die externe Regelnummer der Wurzelregel des Ableitungsbaums (also der ersten Regel der Ableitung). Wenn sich aus dem Zielnonterminal der durch `state` beschriebenen Baum nicht ableiten läßt (das ist normalerweise ein Bug), liefert `burm_rule` 0.

Wie kommt man nun zu den Zahlen für andere Nonterminale als das Start-Nonterminal? Dafür stellt der erzeugte Baumparser folgendes Array zur Verfügung:

```
extern short *burm_nts[] = { ... };
```

Dieses Array enthält für jede externe Regel einen Vektor mit den Zahlen für die Nonterminale im Baummuster der Regel. Die Nonterminale einer Regel werden dabei von links nach rechts durchnummeriert. Beispiel:

```
addr:   Plus(con,Mul(Four,reg)) = 5 (0);
```

Dann findet man die Zahl für `con` in `burm_nts[5][0]` und für `reg` in `burm_nts[5][1]`; 5 ist dabei die externe Regelnummer.

Das würde schon reichen, um einen Reducer zu schreiben. Etwas bequemer wird es durch:

```
extern NODEPTR_TYPE *burm_kids(NODEPTR_TYPE p, int eruleno,
                               NODEPTR_TYPE kids[]);
```

Die Parameter sind die Adresse `p` eines Baumes, eine externe Regelnummer, und ein leerer Vektor von Zeigern auf Bäume. Die Prozedur nimmt an, daß p mit der gegebenen Regel abgeleitet werden kann und füllt den Vektor mit den Unterbäumen von `p`, die den Nonterminalen im Baummuster der Regel entsprechen. `burm_kids` liefert `kids`. Für den Baum

```
Plus(Four,Mul(Four,Fetch(Constant)))
```

und die obige Regel würde `burm_kids` in `kids[0] Four` ablegen und in `kids[1] Fetch(Constant)` (in der Form eines Zeigers auf den Fetch-Knoten).

Der Beispielparser in Abb.3 zeigt wie Labeller und Reducer als User-Code implementiert werden können. Der Reducer gibt den Maschinencode für den Subject Tree aus. (Die Routine `burm_string` wird in Kap.3.3 erklärt.)

```
parse(NODEPTR_TYPE p)
{
  burm_label(p);  /* label the tree */
  reduce(p, 1);   /* and reduce it  */
}

reduce(NODEPTR_TYPE p, int goalnt)
{
  int eruleno;
  short *nts;
  NODEPTR_TYPE kids[100];
  int i;

  eruleno=burm_rule(STATE_LABEL(p), goalnt); /* matching rule number */
  nts=burm_nts[eruleno];               /* subtree goal non-terminals */

  if (eruleno==0) {
    fprintf(stderr, "tree cannot be derived from start symbol");
    exit(1);
  }

  burm_kids (p, eruleno, kids); /* initialize subtree pointers     */
  for (i=0; nts[i]; i++)        /* traverse subtrees left-to-right */
    reduce(kids[i], nts[i]);    /* reduce kids                     */

#if DEBUG
  printf ("%s\n", burm_string[eruleno]);          /* display rule */
#endif

  switch (eruleno){
    case 1:
      printf(...);  /* machine-code for rule 1 */
      break;
    case 2:
      printf(...);  /* machine-code for rule 2 */
      break;
    case 3:
    .
    .
    .
  }
}
```

Abbildung 3: Ein Beispiel-Parser

3.3 Weitere Routinen im erzeugten Baumparser

Dieser Abschnitt ist nur für besonders Interessierte.

Mit den bisher behandelten Routinen können bereits vollwertige Parser implementiert werden. Für Diagnose-Zwecke und fortgeschrittene Programmierung ür stellt `iburg` dem Programmierer jedoch noch eine Reihe weiterer externer Funktionen und Makros zur Verfügung.

Für jedes Nonterminal **X** definiert `iburg` eine Preprozessor-Direktive in der Form `burm_X_NT`, die der Numerierung der Nonterminale entspricht. Außerdem wird ein Makro `burm_X_rule(a)` definiert, das dem Funktionsaufruf `burm_rule(a, X)` entspricht. Für die Grammatik in Abb. 1 definiert `iburg`

```
#define burm_reg_NT 1
#define burm_con_NT 2
#define burm_addr_NT 3
#define burm_reg_rule(a) ...
#define burm_con_rule(a) ...
#define burm_addr_rule(a) ...
```

Diese Definitionen sind nur im Code nach dem zweiten "`%%`" sichtbar.

Die `iburg`-Option -I erzeugt Funktionen zur Beschreibung der Eingabe, welche für Diagnostik-Zwecke nützlich sind: Die Vektoren

```
extern char *burm_opname[];
extern short burm_arity[];
```

beinhalten den Namen und die Anzahl der Kinder jedes Terminales. Der Index ist wieder die externe Symbolnummer des Terminales.

Die Vektoren

```
extern char *burm_string[];
extern short burm_cost[][4];
```

beinhalten den Text und den Kostenvektor für jede Regel. Der Index ist die externe Regelnummer der Regel.

Der 0-terminierte Vektor

```
extern char *burm_ntname[];
```

hat den Index von `burm_X_NT` und beinhaltet den Namen des Non-Terminals X.

Die Routinen

```
extern int burm_op_label(NODEPTR_TYPE p);
extern STATEPTR_TYPE burm_state_label(NODEPTR_TYPE p);
extern NODEPTR_TYPE burm_child(NODEPTR_TYPE p, int index);
```

sind die Funktionsversionen der Konfigurationsmakros.

`burm_child(p, 0)` entspricht `LEFT_CHILD(p)`,

`burm_child(p, 1)` entspricht `RIGHT_CHILD(p)`.

Ein Beispiel für deren Anwendung ist der Grammatik-unabhängige Ausdruck

`burm_opname[burm_op_label(p)]`,

der den textuellen Namen des Operators im Knoten, auf den p zeigt, liefert.

4 Aufrufparameter von `iburg`

`iburg` liest eine Baumgrammatik und generiert einen Baumparser in C. Der erzeugte Baumparser kann für sich selbst compiliert werden oder in ein anderes File inkludiert werden. Das Kommando

```
iburg [Optionen] [inputfile [outputfile]]
```

startet `iburg`. Wenn unter `inputfile` eine Datei angegeben wird, erwartet `iburg` seine Input-Grammatik von dort, andernfalls liest es vom Standard Input.

Optionen (Auszug)

`-p prefix`	exportierte Namen mit prefix beginnen (Default ist burm)
`-I`	Erzeugt die Funktionen burm_arity, burm_child, burm_cost, burm_ntname, burm_op_label, burm_opname, burm_state_label und burm_string
`-T`	Tracing (siehe man-Page)

5 BFE (iburg Front End)

Dieses Werkzeug erzeugt automatisch einen Reducer (ähnlich dem in Abb. 3). Die Eingabe ist eine abgewandelte `iburg`-Spezifikation: der Teil vor dem ersten "`%%`" wird 1:1 übernommen; der Regelteil ist jedoch hier zeilenorientiert, wobei die Zeilen folgende Form haben:

```
rule [ '#' Integer { ',' Integer } [ '#' Aktionen ] ]
```

Regeln werden wie in `iburg` geschrieben, die Integers stellen den Kostenvektor dar. Die Definition der externen Regelnummer fällt weg. Die Aktionen bestehen aus C-Code, den der erzeugte Reducer beim Matching mit der jeweiligen Regel ausführen soll; dabei werden die Aktionen der Blätter vor denen der Wurzel ausgeführt.[1]

Der Reducer durchläuft den Ableitungsbaum, nicht (direkt) den Eingabebaum. Allerdings wird dabei doch auf Knoten des Eingabebaums zugegriffen (insbesondere auf `STATE_LABEL(p)`), und zwar für jeden Knoten des Ableitungsbaumes auf den Wurzelknoten des zugehörigen Eingabebaums; d.h., Knoten des Eingabebaums können mehrmals vorkommen (im Fall von Kettenregeln), aber auch gar nicht.

In den Aktionen kann auf den entsprechenden Knoten des Eingabebaumes über `bnode` (`NODEPTR_TYPE`) zugegriffen werden; auf die Bäume, die den in der rechten Seite der Regel vorkommenden Nonterminalen entsprechen, kann mittels `kids[0]` (`NODEPTR_TYPE`), etc. zugegriffen werden. (Die Nonterminale sind von links nach rechts durchnumeriert.)

Beispiel:

```
reg: NOT(reg)     #1# freereg(kids[0]->reg); bnode->reg=newreg(); printf("not $%d,$%d", kids[0]->reg, bnode->reg);
reg: ADD(reg,val) #1# freereg(kids[0]->reg); bnode->reg=newreg(); printf("addq $%d,%d,$%d", kids[0]->reg, kids[1]->val, bnode->reg);
```

Die Aktionen erzeugen Alpha-Assemblercode. Dabei bedienen sie sich einer einfachen Registerbelegung (`newreg()`, `freereg()`). Das Register wird im Feld `reg` des Eingabeknoten abgelegt.

Beachten Sie, dass ein Eingabeknoten mehrmals im Reducer vorkommen kann (wenn es Kettenregeln gibt); um dabei Kollisionen zu vermeiden, sollten die Aktionen von Regeln mit verschiedenen linken Seiten auf verschiedene Felder schreiben.

Der erzeugte Reducer hat folgenden Prototype:

[1] Leider müssen Sie hier auf den Luxus einer attributierten Grammatik verzichten und die Ausführungsreihenfolge beachten.

```
void burm_reduce(NODEPTR_TYPE bnode, int goalnt);
```

wobei `goalnt` für das Startnonterminal 1 ist. Für ein vollständiges Parsen eines Baumes `p` samt Aktionen braucht man also nur folgende Aufrufe zu tätigen:

```
burm_label(p);       /* label the tree */
burm_reduce(p, 1);   /* and reduce it  */
```

Die Ausgabe von `bfe` besteht aus der um den Reducer erweiterten `iburg`-Eingabe und erfolgt auf der Standardausgabe. Der Aufruf erfolgt in der Form

```
bfe [file]
```

Die Ausgabe kann gleich an `iburg` weitergeleitet werden:

```
bfe file.bfe | iburg
```

6 Erstellen von Baumgrammatiken

Stellen Sie zunächst sicher, daß Ihre Grammatik alle möglichen Eingabebäume ableiten kann. In den meisten Fällen wird es ein spezielles Nonterminal (z.B. `reg`) geben, auf das sich alle Bäume und Unterbäume reduzieren lassen. Dann brauchen Sie zunächst nur für jeden Operator eine Regel der Form

```
reg: OP2(reg,reg) = 1 (1); /* bei binaeren Operatoren */
reg: OP1(reg)     = 2 (1); /* bei unaeren Operatoren */
reg: OP0          = 3 (1); /* bei nullaeren Operatoren */
```

Die Kosten können Sie dabei entsprechend den Kosten des erzeugten Codes anders festlegen.

Eine solche Baumgrammatik ist dann auf jeden Fall vollständig (kann also alle Bäume ableiten). Sie können dann beliebig weitere Regeln (mit besseren Baummustern) dazufügen, die Grammatik wird vollständig bleiben.

7 Zusammenarbeit mit den anderen Werkzeugen

`iburg` hat keine spezielle Unterstützung für die Zusammenarbeit mit `lex`, `yacc` oder `ox`.

Allgemein können Sie so vorgehen, dass sie einen Operatorbaum für jedes Statement in einem Attribut des Statements aufbauen. Schließlich verwenden Sie ein *parse-tree traversal*, um den Baum in einer bestimmten Reihenfolge abzuklappern, und rufen dabei für jedes Statement und seinen Baum den Labeller und den Reducer auf, wobei der Reducer den erzeugten Code ausgibt. Bei komplizierteren Statements (wie z.B. dem IF-Statement) ist es eventuell nicht so sinnvoll, einen Baum für das ganze Statement zu bauen, sondern statt dessen mehrere Bäume für Unterausdrücke und Unterstatements.

Die Registerbelegung führen Sie am besten während des Reducer-Durchgangs durch.

In den Übungsbeispielen können Sie meistens die Operatoren der Quellsprache 1:1 als Operatoren der Zwischencode-Bäume für die Codeauswahl übernehmen; in realistischen Programmiersprachen dagegen wird öfters ein Quellcode-Konstrukt (z.B. Arrayzugriff) in mehrere Zwischencode-Operatoren (z.B. Multiplikation, Addition, und Speicherzugriff) expandiert.

8 Literatur

[FHP92] ist das ursprüngliche `burg`-Manual. [Pro95] beschreibt, wie `burg` Baumautomaten erzeugt und diskutiert weitere Anwendungen von Baumparsern neben der Befehlsauswahl. [FHP93] beschreibt, wie die von `iburg` und die von ihm erzeugten Parser funktionieren.

Literatur

[FHP92] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992. Available from ftp://kaese.cs.wisc.edu/pub/burg.shar.Z.

[FHP93] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1993. Available from ftp://ftp.cs.princeton.edu/pub/iburg.tar.Z.

[Pro95] Todd A. Proebsting. Burs automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.