BISON - Einleitung

- `bison` ist ein Programmgenerator, welches ein C oder C++ Programm erzeugt, dass die syntaktische Analyse eines Tokenstroms durchführt
- Vorgehensweise bie der Erstellung eines Parsers mit `flex` und `bison`:
  - Flex
    - Spezifikation lexikalischer Einheiten (Token) als reguläre Ausdrücke in einer Lexer-Spezifikationsdatei (z.B. `tiger.lex`)
    - Übersetzen nach `lex.yy.c`
  - Bison
    - Spezifikation der syntaktischen Struktur als kontextfreie Grammatik in einer Bison-Spezifikationsdatei (z.B. `tiger.y`)
    - Einbinden des Scanners in den Parser
    - Übersetzen nach `tiger.tab.c`

Bision-Parser Spezifikation

- **Struktur**

  Definitionsteil

  %%

  Regelteil

  %%

  Benutzerdefinierte Routinen

- **Definitionsteil**
  - Funktion des Definitionsteils:
    - Definition von Token
    - Festlegen des Startsymbols
    - Einbinden von C- bzw. C++-Code
    - Festlegung von Prioritäts- und Assoziationsbedingungen
    - Definition von beliebigen Rückgabetypen

Definitionsteil

- **Definition von Token**
  - flex liefert bison die Token als Integer-Werte; genauer lex.yy.c enthält die Funktion yylex(), die die Integer-Werte der erkannten Token an den Parser übergibt
  - Vergabe von Int-Werten an Token
    - Zum konsistenten Gebrauch der Tokenkodierung in flex und bison gelten folgende Konventionen:
      - Token ist ein einzelnes Zeichen (Literal) → ASCII-Wert wird verwendet (1-256)
      - Token ist eine Zeichenkette, die von flex durch einen regulärer Ausdruck erkannt wurde:
        - Es wird ein Tokenname in bison definiert und entsprechende symbolische Konstanten erzeugt (≥ 257)
        - Die symbolischen Konstanten werden flex zugänglich gemacht und dort als Rückgabewert verwendet
      - Token ≤0 signalisieren Ende des Eingabetextes

Definitionsteil (cont.)

- **Tokendefinitionen in `bison`**

```
%token name1, name2, ..., namen

oder

%token name1
%token name2
...
%token namen
```

- Gültige Namen für Token sind alle C-Identifier
- Konvention: Tokennamen werden in Großbuchstaben geschrieben

Definitionsteil (cont.)

- **Generierung der symbolischen Konstanten**
  - Die Option –d beim Aufruf von `bision` erzeugt zusätzlich zu `y.tab.c` die Datei `y.tab.h`, welche C-Konstantendefinitionen für die definierten Token enthält
- **Verwendung in der Lexer-Spezifikationsdatei**
  - Die Datei `y.tab.h` wird im Definitionsteil der Lex-Spezifikationsdatei eingebunden
  - Die Aktionen zu den regulären Ausdrücken enden mit Return-Anweisungen, die als Rückgabewert die symbolische Konstante für das erkannte Token enthalten
- **Beispiel**
  - Bei der Verarbeitung eines Eingabestroms durch den Parser `numId.y` sollen für ganze Zahlen und C-Bezeichner die Token `NUMBER` und `ID` verwendet werden
    - Definition der Token in `numId.y`

Definitionsteil (cont.)

```
%token NUMBER ID
%%
exps:   exp ',' exps
        ;
exps:   exp
        ;
exp:    NUMBER
        | ID
        ;
%%
```

- Gerierung symbolischer Konstanten

  `bison –d numId.y`

- Inhalt der Header-Datei `numId.tab.h`

```
#define NUMBER 257
#define ID 258
```

Definitionsteil (cont.)

- Verwendung der symbolischen Konstanten in der Lex-Spezifikationsdatei `numId.y`

```
%{ #include "numId.tab.h" %}
%%
","                         {printf("%d\n", *yytext);
                             return(',');}
[+-]?(0|[1-9][1-9]*)        {printf("%d\n", NUMBER);
                             return(NUMBER);}
[a-zA-Z_][a-zA-Z0-9_]*      {printf("%d\n", ID);
                             return(ID);}
[\n\t\f\ ]                  {printf("%d\n", '\n');
                             continue;}
.                           {printf("wrong token
                             detected: %c\n", *yytext) ;
                             continue;}
```

Definitionsteil (cont.)

- **Festlegen des Startsymbols**
  - Das Startsymbol wird folgendermaßen festgelegt:

```
%start nonterminal
```

  - Definition nicht zwingend, aber empfohlen für bessere Lesbarkeit
  - Falls kein Startsymbol definiert wurde, wird defaultmäßig die linke Seite der ersten Regel als Startsymbol interpretiert
- **Einbinden von C- bzw. C++-Code**
  - Analog zur Lexer-Spezifikation

```
%{
   C- bzw. C++-Anweisungen
%}
```

Definitionsteil (cont.)

- **Prioritäts- und Assoziativitätsbedingungen**
- **Assoziativität**
  - Assoziativität wird durch die Befehle `%left` und `%right` gewährleistet
  - Mit den Anweisungen

    `%left` Token_1, Token_2, ..., Token_n

  - oder

    `%left` Token_1
    `%left` Token_2
    ...
    `%left` Token_n

  - lassen sich Token im Definitionsteil als linksassoziative definieren
  - Analog werden mit `%right` rechtsassoziative Token definiert

Definitionsteil (cont.)

- **Prioritäten**
  - Die Reihenfolge der `%left` und `%right` Angaben bestimmen die Prioritäten der einzelnen Token
  - Die Priorität der Angaben nimmt von oben nach unten zu
  - Beispiel:
  - Die folgenden Anweisungen

```
%left '+' '-'
%left '*' '/'
```

  - legen fest, dass
    - `'+' '-' '*' '/'` linksassoziative sind
    - `'*' '/'` höhere Priorität als '+' und '-' haben
    - `'*'` und `'/'` sowie `'+'` und `'-'` jeweils gleiche Priorität haben

Definitionsteil (cont.)

- **Prioritäten (cont.)**
  - Mit der Anweisung

    `%prec Token`
  - auf der rechten Seite einer Regel lässt sich dieser die Priorität des angegeben Tokens zuweisen
  - Beispiel

```
...
%left ‘+’ ‘-’
%left ‘*’ ‘/’
%left VORZEICHEN
%%
exp:      ...
        | exp ‘-’ exp
        ...
        | ‘-’ exp %prec VORZEICHEN
...
%%
```

Definitionsteil (cont.)

- **Beliebige Rückgabetypen**
  - Der Standard-Datentyp für den Rückgabewert von Aktionen (`$$, $1, ...`) oder des einem Token zugeordneten Wertes (`yylval`) ist int.
  - Sollen weitere Datentypen für die Rückgabe verwendet werden, so müssen diese mittels des `%union` Schlüsselwortes im Definitionsteil festgelegt werden.:

```
%union{
    typ1 typname1
    typ2 typname2
    ...}
```

- **Anwendung bei Token**
  - Es ist zu kennzeichnen, welchen Rückgabedatentyp ein Token haben soll:

```
%token <typnamei> terminal
```

- **Anwendung bei Nichtterminalen**
  - Für Nichtterminale wird mittels des `%type` Schlüsselwortes der Rückgabewert festgelegt

```
%type <typnamei> nichtterminal
```

Regelteil

- Funktion des Regelteils
  - Festlegung der Nichtterminalen (geschieht implizit)
  - Spezifikation der Produktionen und der dazugehörigen Regeln

- Aufbau des Regelteils
  - Eine Produktion

$$
\begin{array}{lll}
A & \rightarrow & X_{11}\ X_{12}\ \ldots\ X_{1n1} \\
 & | & X_{21}\ X_{22}\ \ldots\ X_{2n2} \\
 & | & \ldots \\
 & | & X_{m1}\ X_{m2}\ \ldots\ X_{mnm}
\end{array}
$$

wobei $A \in$ Nichtterminale, $X_{ij} \in$ Nichtterminale $\cup$ Terminale

  - wird als bison-Regel in der folgenden Form angegeben:

Regelteil (cont.)

```
lsymbol   :     rsymbol11 {aktion11} ... rsymbol1n1 {aktion1n1}
          |     rsymbol21 {aktion21} ... rsymbol2n2 {aktion2n2}
          |     ...
          |     rsymbolm1 {aktionm1} ... rsymbolmnm {aktionmnm}
          ;
```

- **Terminalsymbole**
  - Token aus Definitionsteil
  - Literale, d.h., einzelne Zeichen in Hochkomma, z.B. `'\n'`
- **Nichtterminalsymbole**
  - implizit definiert durch
    - linke Seite einer Regel
    - alle Symbole der rechten Seite, welche nicht Terminalsymbol sind
  - Jedes vorkommende Nichtterminalsymbol muss auf der linken Seite von mindestens einer Regel stehen
  - Konvention: Nichtterminalsymbole werden im Gegensatz zu Terminalsymbolen klein geschrieben

Regelteil (cont.)

- **ε-Regeln**
  - ε-Regeln werden durch eine leere rechte Seite realisiert

```
lsymbol  :       /* leer */
         |       rsymbol1 rsymbol2
         ;
```

  - oder explizit

```
lsymbol  :       epsilon
         |       rsymbol1 rsymbol2
         ;
epsilon  :       /* leer */
         ;
```

Regelteil (cont.)

- **Aktionen**
  - Zu jedem Symbol auf der rechten Seite kann eine zugehörige Aktion angegeben werden
  - Eine Aktion besteht aus einer oder mehreren C- bzw. C++-Anweisungen in geschweifter Klammer
  - Eine Aktion $\texttt{aktion}_{ij}$ wird dann ausgeführt, wenn das Symbol $\texttt{rsymbol}_{ij}$ abgedeckt worden ist.
  - Aktionen, die nicht am Ende einer Regel angegeben sind, werden wie ein nichtterminales Symbol behandelt
  - Es wird intern eine zusätzliche Regel eingefügt, welche als linke Seite dieses Nichtterminalsymbol hat und als rechte Seite nur die Aktion selbst
- **Binden von Werten an Symbole**
  - Es ist möglich, Werte an Terminal- und Nichtterminalsymbole zu binden
  - Defaultmäßig sind dies int-Werte

Regelteil (cont.)

- **Verwenden von Werten**
  - Werte der Symbole einer rechten Seite

    $rsymbol_1$ ... $rsymbol_n$

  - sind über die Variablen

    $1 ... $n

  - verfügbar.
  - N.B.: Aktionen, die nicht am Ende einer Regel stehen, werden auch als Symbol behandelt
- **Zuweisung von Werten an Nichtterminalsymbole**
  - Zuweisung eines Wertes an die linke Seite einer Regel
  - Verwendung des Platzhalters $$ für die Zuweisung; z.B. Zuweisung an Nichtterminal a:

```
a :        b ‘+’ c {$$ = $1 + $3;}
  ;
```

Regelteil (cont.)

- Defaultmässige Zuweisung des ersten Wertes der rechten Seite an $$:

```
a          :        b '+' c {$$ = $1;} /* default */
           ;
```

- Zuweisung von Werten an Terminalsymbole
  - Verwendung der gemeinsamen Variablen `yylval` in `flex` und `bison`
  - Zuweisung des Wertes eines Token an `yylval` während des Scannens
- Beispiel:

```
/* dualToDecimal.lex */
%{
#include "dualToDecimal.tab.h"
%}
%%
[01]    {yylval=*yytext-'0'; return(DNUMBER);}
\n      return(*yytext);
.       ;
%%
```

Regelteil (cont.)

```
/* dualToDecimal.y */
%{
 #include <stdio.h>
  void yyerror (const char *); %}
%token  DNUMBER '\n'
%%
values    :       number  {printf("%d\n"), $1;} '\n'
          |       values  number  {printf("%d\n", $2);} '\n'
          ;
number    :       /* empty */     {$$=0;}
          |       number  {$$=$1*2;}      DNUMBER {$$=$2+$3;}
          ;
%%
void yyerror (const char *s) {
     fprintf (stderr, "parse error: %s\n", s);}
int main(int argc, char *argv[]) {
     yyparse();
     return(0); }
```

Regelteil (cont.)

- **Fehlerbehandlung**
- Default Fehlerbehandlung
  - Beim Lesen eines Tokens, welches nicht in die syntaktische Struktur paßt:
    - Ausgabe der Meldung “`syntax error`”
    - Abbruch
  - Wünschenswert ist eine Fehlerbehandlung, die
    - präzisere Informationen über einen aufgetretenen Fehler liefert (Grund, Zeilennummer, ...)
    - mit der Syntakanalyse an einem geeigneten Punkt fortfährt
- **Das Hilfssymbol `error`**
- bison stellt das “künstliche” Nichtterminal `error` zur Verfügung
- Verwendung bei fehlergefährdeten Produktionen als alternative rechte Seite
- Zusätzlich Angabe eines Aufsatztokens möglich

Regelteil (cont.)

- Beispiel:

```
exps :          exp
       |        exps exp
       |        error ';' {froutine()}
       ;
```

- Funktionsweise:
  - Fehler beim Lesen eines Tokens → Fehlerbehandlungsmodus wird eingeschalten
  - Solange Zustande von Stack entfernen, bis Zustand erreicht ist, in dem der Punkt vor Nichtterminal error steht
  - Falls solch ein Zustand nicht existiert → Ausgabe von "syntax error" und Abbruch
  - Solange Eingabezeichen lesen, bis Aufsetzzeichen erkannt wird
  - Fehlerbehandlungsmodus wird verlassen, sobald drei gültige Eingabesymbole erkannt werden
  - Direktes Verlassen durch Aufruf der Funktion `yyerrok()` möglich

Regelteil (cont.)

- **Die Funktion `yyerror`**
  - Funktion
    `void yyerror(char *s)`
  - is zuständig für die Ausgabe der Fehlermeldung
  - Anpassung der Fehlermeldung durch Überschreiben der Funktion möglich
- **Die `char`-Variable `yychar`**
  - enthält die Tokennummer des aktuellen look-ahead Tokens

Benutzerdefinierte Routinen

- Funktion
  - Code wird unverändert nach `y.tab.c` kopiert
  - Einbinden von benutzerdefinierten Funktionen, die in den Aktionen benutzt werden können
  - Bereitstellung der Funktion `int yylex()`
  - ggf. Überschreiben der Funktion `yyerror()` und `int main()`

Konflikte und Mehrdeutigkeiten

- **Konflikte**
  - bison verarbeitet LALR(1)-Grammatiken
  - Ist die gegebene Grammatik nicht LALR(1), so meldet bison einen Konlfikt:
    - shift/reduce Konflikt
    - reduce/reduce Konflikt
  - Im Konfliktfall ist die Generierung eines Aktion- und Goto Tabelle hilfreich
  - Dazu steht die Option –v für bison zur Verführung
  - Konflikte lassen sich bei Anwendung der Option direkt aus der Tabelle y.output ablesen
- **Default Konfliktlösung**
  - Konflikte werden von bison nach folgender Regel aufgelöst:
    - shift/reduce Konflikt $\rightarrow$ es wird shift angewendet
    - reduce/reduce Konflikt $\rightarrow$ es wird erstgegebene Regel im Regelteil zur Reduktion verwendet
  - Besser: Konflikte selbt auflösen durch
    - Transformation der Grammatik oder
    - Festlegung von Priotitäts- und Assoziativitätsbedingungen