Programming Paradigms and Formal Semantics

Lab 1: Definite Clause Grammars

© Vadim Zaytsev, Ralf Lämmel
(@grammarware & @notquiteabba),
Software Languages Team,
Universität Koblenz-Landau

Lab structure

- All work is done in teams
- Each team has 3 students
- Each lab introduces new assignment
  - assignment PDF will also appear on-line
  - topic explanations happen at the lab
- Each team submits a solution each time
  - where: subversion repository
  - deadline: end of Sunday
- Each lab starts with solution demonstrations
  - several teams are called at random
  - each good solution presented earns a point
- Each team needs 2 points before midterm

Teams so far

- ebp
  - Eiserloh, Becker, Peifer
- edistefjo
  - Hartung, Schleining, Härtel
- group1
  - Wolf, Normann?, Zitz
- Jigsaw
  - Brenk, Hück, Klauer
- ssp
  - Saal, Schmorleiz, Polster
- beck
  - Baltzer, Barjenbruch, Rossberg
- miregal
  - Humbert, Kornas, Merz

Prolog

Horn clauses

$\neg t_1 \vee \neg t_2 \vee \neg t_3 \vee p$

$t_1 \wedge t_2 \wedge t_3 \rightarrow p$

$p \leftarrow t_1 \wedge t_2 \wedge t_3$

In Prolog notation

```
p :- t1, t2, t3.
```

Facts

```
fact(par).
fact(par) :- true.
```

Predicates

$pred_1(par_1,par_2)$:-

    $pred_2(par_1,par_3)$,

    $pred_3(par_2,par_3)$.

DCG

Horn clauses have at most one positive literals.
Definite clauses have exactly one positive literal.
Approach by Fernando Pereira and David Warren (1980).
Alan Colmerauer's metamorphosis grammars (1978).
Predecessors/related approaches:
    attribute grammars (Donald E. Knuth, 1968)
    van Wijngaarden grammars (Adriaan v.W., 1969)
In EBNF:
    program ::= statements;
    statements ::= statement;
    statements ::= statement ";" statements;
In DCG:
    program --> statements.
    statements --> statement.
    statements --> statement, terminal(";"), statements.

DCG in Prolog

Implicit parameters: difference lists

Nonterminals can be explicitly parametrised:

```
s --> a(N), b(N), c(N).
a(0) --> [].
a(M) --> [a], a(N), {M is N + 1}.
b(0) --> [].
b(M) --> [b], b(N), {M is N + 1}.
c(0) --> [].
c(M) --> [c], c(N), {M is N + 1}.
```

$\{a^n b^n c^n\}$ is not a context-free language

While

n ::= (number)
x ::= (identifier)
$a ::= n \mid x$
$\quad \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$
$b ::= true \mid false$
$\quad \mid a_1 = a_2 \mid a_1 \leq a_2$
$\quad \mid \neg b \mid b_1 \wedge b_2$
$S ::= x := a \mid skip \mid S_1 ; S_2$
$\quad \mid$ if b then S_1 else S_2
$\quad \mid$ while b do S

(Lecture1 deck, slide 22)

(Semantic with Applications, page 7)

While: program

```
% Program is a list of statements
program(S) --> statements(S).

% Non-empty list
statements(slist(H,T)) -->
    statement(H),
    keyword(";"),
    statements(T).

% Empty list
statements(S) --> statement(S).
```

While: statements

```
% Statements
% Skip statement
statement(skip) -->
    keyword("skip").

% Assign statement
statement(assign(identifier(V),E)) -->
    identifier(V),
    keyword(":="),
    aexpression(E).
```

While: statements

```
% Conditional statement
statement(ifthenelse(E,S1,S2)) -->
    keyword("if"),
    bexpression(E),
    keyword("then"),
    statement(S1),
    keyword("else"),
    statement(S2).
```

While: arithmetic expressions

```
% Number is a an arithmetic expression
aexpression(number(N)) --> number(N).

% Variable reference
aexpression(identifier(V)) --> identifier(V).

% Sum of arithmetic expressions
aexpression(add(E1,E2)) -->
    keyword("("),
    aexpression(E1),
    keyword("+"),
    aexpression(E2),
    keyword(")").
```

While: boolean expressions

```
bexpression(true) --> keyword("true").
bexpression(false) --> keyword("false").

bexpression(equals(E1,E2)) -->
 aexpression(E1),
 keyword("="),
 aexpression(E2).

bexpression(lte(E1,E2)) -->
 aexpression(E1),
 keyword("≤"),
 aexpression(E2).
```

While: low-level bits

```
% Dealing with spaces
spaces --> [0' ], spaces.
spaces --> [].

% Keywords are space-consuming strings
keyword(X) -->
    spaces,
    string(X).

% Bare strings
string([],X,X).
string([H|T1],[H|T2],X) :- string(T1,T2,X).
```

While: low-level bits

```
% Numbers are space-consuming digits
number(N) -->
    spaces,
    digit(H), digits(T),
    { number_chars(N,[H|T]) }.

% Bare digits
digits([H|T]) --> digit(H), digits(T).
digits([]) --> [].
digit(H,[H|T],T) :- H >= 0'0, H =< 0'9.

identifier(V) -->
    …
```

While

$n ::= (number)$

$x ::= (identifier)$

$a ::= n \mid x$
$\quad \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$

$b ::= true \mid false$
$\quad \mid a_1 = a_2 \mid a_1 \leq a_2$
$\quad \mid \neg b \mid b_1 \wedge b_2$

$S ::= x := a \mid skip \mid S_1 ; S_2$
$\quad \mid if\ b\ then\ S_1\ else\ S_2$
$\quad \mid while\ b\ do\ S$

Assignment: finish it!

XML

The core of XML-like syntax with XML-like semantics (trees).
No textual content, no attributes, no namespaces, …

```
<x>
    <y></y>
</x>
```

In EBNF:

```
tree ::= "<" tag ">" tree* "</" tag ">";
         ^^^                ^^^
```

XML

```
% A tree has opening and closing tags and
% a forest in between
tree(tree(N,L)) -->
 spaces,
 string("<"),
 tag(N),
 string(">"),
 trees(L),
 string("</"),
 tag(N),
 string(">"),
 spaces.
```

XML

```
% A non-empty forest
trees([H|T]) -->
 tree(H),
 trees(T).

% An empty forest
trees([]) --> [].

% Only lowcased tag names are allowed
tag(N) -->
 letter(H), letters(T),
 { atom_codes(N,[H|T]) }.
```

XML

Add attributes:

```
<x a='2'>
    <y b='true' c='no'></y>
</x>
```

In EBNF:

```
tree ::= "<" tag attrs?">" tree* "</" tag ">";
              ^^^                  ^^^
attrs ::= (aname "='" value "'")+;
           ^^^^^^
```

Assignment: finish it!

Conclusion

- Send team info if not already done so
  - Submit everything on time
- Show up at least two times with decent solutions

- Each lab will have several teams presenting
- Each lab will introduce a new topic
- Each lab will explain a new assignment

- Questions can be tweeted to @grammarware or @notquiteabba or @yaplcourse or emailed
- The working code/tests/makefiles are available via SLPS: http://slps.sf.net, use SVN to check out.