Meta Programming with Concrete Object Syntax

Master Course Program Transformation 2004-2005

Martin Bravenboer

Institute of Information & Computing Sciences
Utrecht University,
The Netherlands

March 8, 2005

Meta-Programming with Concrete Object Syntax

- *Meta*-programming:
  - Generating, transforming, or analyzing *object* programs
- Meta-programs should use concrete syntax of object language
  - Implications for meta-language implementation
- General architecture for extending meta-languages
  - Modular syntax definition
  - Meta explosion and assimilation
  - By default available in Stratego/XT
- Running Example
  - Meta language: Stratego
  - Object language: Tiger
  - Meta program: instrumentation of Tiger program

Case Study: Instrumenting Programs

```
let function fact(n : int) : int =
      if (n < 1)
      then 1
      else n * fact(n - 1)
in printint(fact(10))
end
```

```
fact entry
   fact entry
      fact entry
         fact entry
           ...
         fact exit
      fact exit
   fact exit
fact exit
```

Meta-program *TraceAll*:

- Instrument a Tiger program to trace function calls
- *Generate* code for `enterfun` and `exitfun`
- *Transform* code for functions

Case Study: Instrumenting Programs

```
let var ind := 0
    function enterfun(name : string) =
      (ind := (ind + 1);
       for n := 2 to ind do print("   ");
       print(name);
       print("  entry"))
    function exitfun(name : string) =
      (for n := 2 to ind do print("   ");
       ind := (ind - 1);
       print(name);
       print("  exit "))
    function fact(n : int) : int =
      (enterfun(" fact");
       let var a_0 : int := nil
       in if (n < 1)
          then a_0 := 1
          else a_0 := n * fact(n - 1);
          exitfun(" fact");
          a_0
       end)
in printint(fact(10))
end
```

Generation with String-Based Concrete Syntax

```
IntroducePrinters :
  e -> "let var ind := 0"
    ++ "    function enterfun(name : string) ="
    ++ "      (ind := +(ind, 1);"
    ++ "       for i := 2 to ind do print(\" \");"
    ++ "       print(name); print(\" entry\"))"
    ++ "    function exitfun(name : string) = "
    ++ "      (for i := 2 to ind do print(\" \");"
    ++ "       ind := -(ind, 1);"
    ++ "       print(name); print(\" exit\"))"
    ++ " in "
    ++ e
    ++ "end"
```

- *Generation* with concrete syntax
- Readable code
- Easy to implement
- No syntactic checks
- *Transformation* requires analysis of program

Transformation of Abstract Syntax Trees

```
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
```

```
regular tree grammar
  productions
    Var  -> Var(Id)
    Exp  -> Call(Var, Exps)
          | Plus(Exp, Exp)
```

- Structured representation
- Works well with small language and small program fragments
- Analysis and generation of program fragments
- Tree structure: matching and building

Transformation of Abstract Syntax Trees

```
rules
  TraceProcedure :
    FunDec(f, xs, NoTp, e) ->
    FunDec(f, xs, NoTp,
           Seq([Call(Var("enterfun"),[String(f)]), e,
                Call(Var("exitfun"),[String(f)])]))
  TraceFunction :
    FunDec(f, xs, Tp(t), e) ->
    FunDec(f, xs, Tp(t),
           Seq([Call(Var("enterfun"),[String(f)]),
                Let([VarDec(x,Tp(t),NilExp)],
                    [Assign(Var(x), e),
                     Call(Var("exitfun"),[String(f)]),
                     Var(x)])]))
    where new => x
```

- Does not scale up to large program fragments
- Requires deep knowledge of syntactic structure
- Does not scale up to large languages (complex syntax)

Write Concrete & Transform Abstract!

```
rules
  TraceProcedure :
    |[ function f(x*) = e ]| ->
    |[ function f(x*) = (enterfun(s); e; exitfun(s)) ]|
    where !f => s

  TraceFunction :
    |[ function f(x*) : tid = e ]| ->
    |[ function f(x*) : tid =
         (enterfun(s);
          let var x : tid := e
           in exitfun(s); x end) ]|
    where new => x ; !f => s
```

- Can be used in transformation *and* generation
- Structured representation (transformation on trees)
- Fragments are syntactically correct
- Requires ‘no’ deep knowledge of syntactic structure
- Scales up to large languages (complex syntax)

Write Concrete & Transform Abstract!

```
rules
  IntroducePrinters :
    e -> |[ let var ind := 0

                function enterfun(name : string) =
                 (ind := +(ind, 1);
                  for i := 2 to ind do print(" ");
                  print(name); print(" entry"))

                function exitfun(name : string) =
                 (for i := 2 to ind do print(" ");
                  ind := -(ind, 1);
                  print(name); print(" exit"))

            in e end ]|
```

- Scales up to large program fragments

Java: Code Generation

Concrete object syntax is not a Stratego specific issue.

Suppose we want to generate:

```
if(propertyChangeListeners == null)
  return;

PropertyChangeEvent event =
  new PropertyChangeEvent(this, fieldName, oldValue, newValue);

for(int c=0; c < propertyChangeListeners.size(); c++) {
  ((PropertyChangeListener)
      propertyChangeListeners.elementAt(c)).propertyChange(event);
}
```

Parameterized by the name of the listeners variable.

(Fragment generated by Castor)

Java: Code Generation using Strings

```
String vName = "propertyChangeListeners";

jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");

jsc.add("PropertyChangeEvent event = new ");
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue, newValue);");

jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Java: Code Generation using Strings

```
String vName = "propertyChangeListeners

jsc.add("if (");
jsc.append(vName);
jsc.append(" == null) return;");

jsc.add("PropertyChangeEvent event = new
jsc.append("PropertyChangeEvent");
jsc.append("(this, fieldName, oldValue,

jsc.add("for (int i = 0; i < ");
jsc.append(vName);
jsc.append(".size(); i++) {");
jsc.indent();
jsc.add("((PropertyChangeListener) ");
jsc.append(vName);
jsc.append(".elementAt(i)).");
jsc.append("propertyChange(event);");
jsc.unindent();
jsc.add("}");
```

Uses the Java syntax: the syntax of the domain.

No syntactic checks of the generated code.

Escaping to the meta language is difficult.

Code generator tries to do some pretty printing.

Further processing of the code is impossible.

Java: Code Generation using Abstract Syntax Trees

```
VariableDeclarationFragment fragment =
    _ast.newVariableDeclarationFragment();
fragment.setName(_ast.newSimpleName("event"));
ClassInstanceCreation newi = _ast.newClassInstanceCreation();
newi.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
List args = newi.arguments();
args.add(_ast.newThisExpression());
args.add(_ast.newSimpleName("fieldName"));
args.add(_ast.newSimpleName("oldValue"));
args.add(_ast.newSimpleName("newValue"));
fragment.setInitializer(newi);
VariableDeclarationStatement vardec =
    _ast.newVariableDeclarationStatement(fragment);
vardec.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent")));
```

Java: Code Generation using Abstract Syntax Trees

Extremely verbose and unclear: 90 lines of code!

```
VariableD                          t =
    _ast.                          ment();
fragment.s                        ("event"));
ClassInstanceCreation newi = _ast.newClassInstanceCreation();
newi.setType(_ast.newSimpleType(
    _ast.newSimpleName("PropertyChangeEvent"
List args = newi.arguments();
args.add(_ast.newThisExpression());
args.add(_ast.newSimpleName("fieldName"));
args.add(_ast.newSimpleName("oldValue"));
args.add(_ast.newSimpleName("newValue"));
fragment.setInitializer(newi);
VariableDeclarationStatement var
    _ast.newVariableDeclarationS
vardec.setType(_ast.newSimpleTyp
                             rty
```

Does not correspond to the structure of the code to be generated.

Code is syntactically checked by host language compiler and further processing is possible.

Don't worry about the layout.

Java: Code Generation using Concrete Syntax

```
String x = "propertyChangeListeners";

List<Statement> stms = |[
  if(x == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, oldValue, newValue);

  for(int c=0; c < x.size(); c++) {
    ((PropertyChangeListener)
      x.elementAt(c)).propertyChange(event);
  }
]|;
```

Java: Code Generation using Concrete Syntax

```
String x = "propertyChangeListeners";

List<Statement> stms = |[
  if(x == null)
    return;

  PropertyChangeEvent event =
    new PropertyChangeEvent(this, fieldName, ol

  for(int c=0; c < x.size(); c++) {
    ((PropertyChangeListener)
      x.elementAt(c)).propertyChange(event);
  }
]|;
```

Uses the syntax of the domain: Java.

Syntax of the generated code is checked and further processing is possible.

Separate pretty-printer: don't worry about the layout.

Support for interaction between the generated code and the meta language.

Implementation: Meta-Programming a la Concrete Syntax

Syntax of Stratego (meta language)
is extended with
syntax of Tiger (object language)

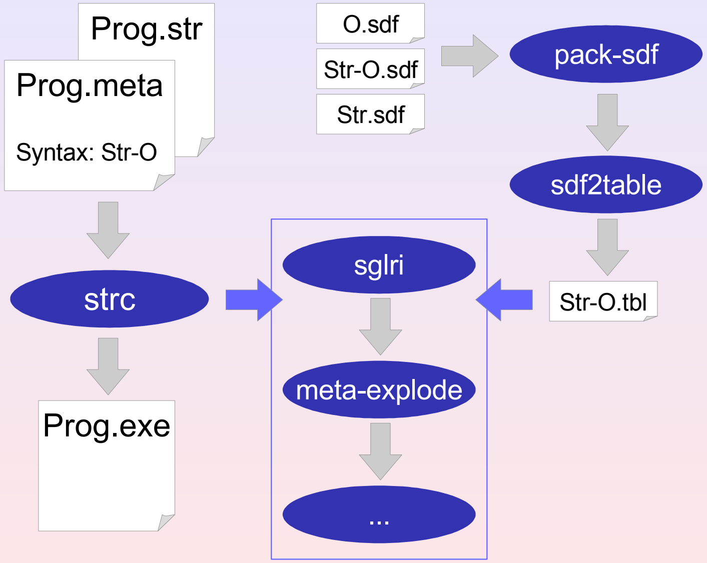
Recipe

1. Combine syntax of meta- and object-languages
2. Parse meta-program with combined syntax
3. Feed parse tree to meta-language compiler

Ingredients

- M-compile: meta-language compiler
- SDF: Modular syntax definition formalism
- SGLR: Scannerless Generalized LR parser

Architecture



How To Use In Stratego (Java)

- Usually, an embedding of `Obj` already exists.
- Use concrete syntax in `Foo.str`

```
module Foo imports Java-15 options
strategies
  main =
    output-wrap(
      !|[ package foo;
          public class Bar {
            public static void main(String[] ps) {
              System.out.println("Hello world!");
            }
          } ]|)
```

- Declare the syntax in `Foo.meta`

```
Meta([ Syntax("Stratego-Java-15") ])
```

- Compile the Stratego program

```
$ strc -i Foo.str -I ...
```

How To Use In Stratego (Tiger)

- Usually, an embedding of `Obj` already exists.
- Use concrete syntax in `Tiger-Profile.str`

```
module Tiger-Profile
imports libstrategolib Tiger

  ...

       InstrumentCall(s) :
         |[ f(a*) ]| -> |[ (z := z + 1; f(a1*)) ]|

  ...
```

- Declare the syntax in `Tiger-Profile.meta`

```
Meta([ Syntax("StrategoTiger") ])
```

- Compile the Stratego program

```
$ strc -i Tiger-Profile.str -I ...
```

Implementation: Syntax of Object Language (Tiger)

SDF Syntax Definition

```
module Tiger
exports
  context-free syntax
    "let" Dec* "in" {Exp ";"}* "end" -> Exp {cons("Let")}
    Id                               -> Var {cons("Var")}
    LValue ":=" Exp                  -> Exp {cons("Assign")}
    "(" {Exp ";"}* ")"               -> Exp {cons("Seq")}
```

Regular Tree Grammar

```
regular tree grammar
  productions
    Exp -> Seq(Exps)
         | Assign(LValue, Exp)
         | Let(Decs, Exps)
    Var -> Var(Id)
```

let *ds* in *x* := (*es*) end $\Rightarrow$ Let(*ds*,[Assign(Var(*x*),Seq(*es*))])

Implementation: Syntax of Meta Language (Stratego)

SDF Syntax Definition of Stratego

```
module Stratego
exports
  context-free syntax
    String                      -> Term {cons("Str")}
    Var                         -> Term {cons("Var")}
    Id "(" {Term ","}* ")" -> Term {cons("Op")}
    Term "->" Term              -> Rule {cons("Rule")}
```

Stratego Fragment, Concrete Syntax

```
Assign(Var(x),Let(ds,es)) -> Let(ds,[Assign(Var(x),Seq(es))])
```

Stratego Fragment, Abstract Syntax

```
Rule(Op("Assign",[Op("Var",[Var("x")]),
                  Op("Let",[Var("ds"),Var("es")])]),
     Op("Let",[Var("ds"),
               Op("Cons",[Op("Assign",[Op("Var",[Var("x")]),
                                       Op("Seq",[Var("es")])]),
                          Op("Nil",[])])]))
```

Implementation: Extending the Meta Language

```
module StrategoTiger
imports Tiger
imports Stratego [ Id   => StrategoId
                   Var  => StrategoVar
                   Term => StrategoTerm ]
exports
  context-free syntax
    "|[" Dec     "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" FunDec  "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" Exp     "]|" -> StrategoTerm   {cons("ToTerm")}

    "~"  StrategoTerm -> Exp            {cons("FromTerm")}
    "~*" StrategoTerm -> {Exp ";"}+     {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id            {prefer}
    [e][0-9]*      -> Exp           {prefer}
    "e"[0-9]* "*"  -> {Exp ";"}+    {prefer}
    "fd"[0-9]* "*" -> FunDec+       {prefer}
```

Implementation: Extending the Meta Language

Rename sorts (non-terminals) to unintended avoid clashes

```
module StrategoTiger
imports Tiger
imports Stratego [ Id   => StrategoId
                   Var  => StrategoVar
                   Term => StrategoTerm ]
exports
  context-free syntax
    "|[" Dec    "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" FunDec "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" Exp    "]|" -> StrategoTerm   {cons("ToTerm")}

    "~"  StrategoTerm -> Exp            {cons("FromTerm")}
    "~*" StrategoTerm -> {Exp ";"}+     {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id           {prefer}
    [e][0-9]*      -> Exp          {prefer}
    "e"[0-9]* "*"  -> {Exp ";"}+   {prefer}
    "fd"[0-9]* "*" -> FunDec+      {prefer}
```

Implementation: Extending the Meta Language

```
module StrategoTiger
imports Tiger
imports Stratego [ Id   => StrategoId
                   Var  => StrategoVar
                   Term => StrategoTerm ]
exports
  context-free syntax
    "|[" Dec    "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" FunDec "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" Exp    "]|" -> StrategoTerm   {cons("ToTerm")}

    "~"  StrategoTerm -> Exp            {cons("FromTerm")}
    "~*" StrategoTerm -> {Exp ";"}+     {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id           {prefer}
    [e][0-9]*      -> Exp          {prefer}
    "e"[0-9]* "*"  -> {Exp ";"}+   {prefer}
    "fd"[0-9]* "*" -> FunDec+      {prefer}
```

Quotation: inject object expressions in meta expressions

Anti-quotation: inject meta expressions into object expressions

Implementation: Extending the Meta Language

```
module StrategoTiger
imports Tiger
imports Stratego [ Id   => StrategoId
                   Var  => StrategoVar
                   Term => StrategoTerm ]
exports
  context-free syntax
    "|[" Dec    "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" FunDec "]|" -> StrategoTerm   {cons("ToTerm")}
    "|[" Exp    "]|" -> StrategoTerm   {cons("ToTerm")}

    "~"  StrategoTerm -> Exp           {cons("FromTerm")}
    "~*" StrategoTerm -> {Exp ";"}+    {cons("FromTerm")}

  variables
    [xyzfgh][0-9]* -> Id          {prefer}
    [e][0-9]*      -> Exp         {prefer}
    "e"[0-9]* "*"  -> {Exp ";"}+  {prefer}
    "fd"[0-9]* "*" -> FunDec+     {prefer}
```

Declare meta-variables for object sorts

Meta Variables

```
TraceProcedure :
  |[ function f(x*) = e ]| ->
  |[ function f(x*) = (enterfun(s); e; exitfun(s)) ]|
  where
    !f => s
```

meta-variables
object identifiers

```
  variables
    [xyzfgh][0-9]* -> Id           {prefer}
    [s][0-9]*      -> StrConstr    {prefer}
    [ijkn][0-9]*   -> IntConst     {prefer}
    [e][0-9]*      -> Exp          {prefer}
    "e"[0-9]* "*"  -> {Exp ";"}+   {prefer}
    "a"[0-9]* "*"  -> {Exp ","}+   {prefer}
    "fd"[0-9]* "*" -> FunDec+      {prefer}
```

Anti-Quotation

```
TraceProcedure :
  |[ function ~f(~*xs) = ~e ]| ->
  |[ function ~f(~*xs) = (enter(~s); ~e; exit(~s)) ]|
  where
    !f => s
```

Distinguish meta-variables by anti-quotation

```
TraceProcedure :
  |[ function ~f(~*xs) = ~e ]| ->
  |[ function ~f(~*xs) =
       (print(~String(<conc-strings>(f," entry")));
        ~e;
        print(~String(<conc-strings>(f," exit")))) ]|
```

Splice code generated by meta-computation into object code

Congruences

```
cong1 = |[ if <s> then <id> else <id> ]|

cong2 = |[ let <*map(s)> in <*id> end ]|

cong3 = |[ let <fd:s> in <*id> end ]|

cong4 = |[ let <fd*:map(s)> in <*id> end ]|
```

Concrete syntax for congruences

```
context-free syntax
  "<"     StrStrategy ">" -> Exp     {cons("FromApp")}
  "<*"    StrStrategy ">" -> Dec*    {cons("FromApp")}
  "<fd:"  StrStrategy ">" -> FunDec  {cons("FromApp")}
  "<fd*:" StrStrategy ">" -> FunDec+ {cons("FromApp")}
```

Exploding Embedded Abstract Syntax: Example

```
|[ x := let ds in ~* es end ]| -> |[ let ds in x := (~* es) end ]|
```

```
Rule(ToTerm(Assign(Var(meta-var("x")),
                        Let(meta-var("ds"),FromTerm(Var("es"))))),
     ToTerm(Let(meta-var("ds"),
                     [Assign(Var(meta-var("x")),
                             Seq(FromTerm(Var("es")))])))
```

```
Rule(Op("Assign",[Op("Var",[Var("x")]),
                        Op("Let",[Var("ds"),Var("es")])]),
     Op("Let",[Var("ds"),
                    Op("Cons",[Op("Assign",[Op("Var",[Var("x")]),
                                            Op("Seq",[Var("es")])]),
                               Op("Nil",[])])]))
```

```
Assign(Var(x),Let(ds,es)) -> Let(ds,[Assign(Var(x),Seq(es))])
```

SGLR: Scannerless Generalized LR Parsing

Restricted classes of context-free grammars not closed under composition.

- LR/LL parsers do not compose
- Scanners (regular grammars) do not compose

Scannerless Generalized LR Parsing

- Integration of lexical and context-free syntax
- Handle full class of context-free grammars
- Context-based disambiguation
- Unbounded lookahead
- Supports *natural* syntax definition
- Limitation: only *context-free* grammars

Applications

- Java code generation and transformation
  - Java-front
- XML generation in Stratego
  - Available in Stratego/XT
- Handcrafted pretty-printers: Box embedded in Stratego
  - Available in Stratego/XT
- Meta Stratego
  - Stratego Compiler
  - Stratego code generators
- Meta SDF
  - Code generators in Stratego/XT

Application: Tiger/XML

```
function webpage(title : elt, body : elt) : elt =
  <html>
    <head>
      <title> $title </title>
    </head>
    <body>
       $body
    </body>
  </html>
```

Embed XML in a programming language

Application: Stratego/XML

```
ra2mathml(s) =
  !%>
    <?xml version="1.0"?>
    <math xmlns="http://www.w3.org/1998/Math/MathML">
      <% s %>
    </math>
  <%

ra-to-presentation-mathml(x) :
  Project(ps, r) ->
    %><mrow>
        <msub>
          <mo>&pi;</mo>
          <mrow>
            <mfenced open="" close="">
              <% <map(x)> ps :: * %>
            </mfenced>
          </mrow>
        </msub>
        <mfenced>
          <% <x> r %>
        </mfenced>
      </mrow><%
```

Application: Stratego/Tiger/XML

```
ExpElt :
  |[ <tag> cont* </tag> ]| ->
  |[ let var x := contents[k] of nil
       in ( e* );
          elt{tag = s , text = "", contents = x , size = k}
      end ]|
  where
    new => x ; !tag => s
  ; <length> [cont*] => k
  ; <dec; upto> k => is
  ; <zip(
    \(cont , i ) -> |[ x[i] := <content> cont </content> ]|\
    )> ([cont*], is ) => [e*]
```

Embedding can be nested

Application: Stratego/Stratego

```
Desugar :
  |[ s => t ]| -> |[ s; ?t ]|

Desugar :
  |[ <s> t :S]| -> |[ !t; s ]|

Desugar :
  |[ f(as) : t1 -> t2 where s ]|
    ->
  |[ f(as) = ?t1; where(s); !t2 ]|

Desugar :
  |[ if s1 then s2 else s3 end ]|
    ->
  |[ where(s1) < s2 + s3 ]|
```

Transformation of Stratego programs in Stratego

Application: Stratego/Box

```
expr-to-box :
  Plus(b1, b2) -> H hs=1 [ b1 "+"  b2]

UglyPrint :
  If(b1, b2, b3) ->
    V vs=0 [
      H hs=0 [KW["if"] "(" b1 ")"]
      b2
      KW["else"] b3]

PrettyPrint :
  If(b1, b2, If(b3, b4, b5)) ->
    V vs=0 [
      H hs=0 [KW["if"] "(" b1 ")"]
      b2
      H hs=1 [KW["else"] H hs=0 [KW["if"] "(" b3 ")"]]
      b4
      KW["else"] b5]
```

Application: Stratego/Java

Next Lecture!

Application: Java/Java

Embed Java syntax in Java

```
context-free syntax
  "type" "|[" Type "]|" -> MetaExpr {cons("ToMetaExpr")}

variables
  "e" [0-9]*     -> Expr          {prefer}
  "e" [0-9]* "*" -> {Expr ","}*   {prefer}
```

Assimilation rules for Eclipse JDT Core API

```
  Assimilate(r) :
    type |[ double ]| -> |[ ast.newPrimitiveType(PrimitiveType.DOUBLE) ]|

  Assimilate(r) :
    |[ y(e*) ]| -> |[
      {| MethodInvocation x = ast.newMethodInvocation();
         x.setName(ast.newSimpleName("~y"));
         bstm* | x |}
    ]|
    where <newname> "inv" => x
        ; <ExplodeArgs(r | x)> e* => bstm*
```

Conclusion

Extend meta language with object language syntax

- combine arbitrary meta and object language
- meta-language does not need to be designed for this purpose
- limitation: language should have context-free syntax

Modular syntax definition is essential

- reuse existing syntax definitions – no changed needed

Scannerless Generalized-LR parsing

- Only context-free grammars closed under composition
- No compositionality: regular grammars, LR, LL