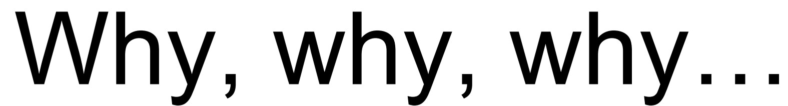
Bernoulli Compiler Construction Kit a.k.a. “The XML stuff”

Kamen Yotov
kamen@yotov.org
Department of Computer Science
Cornell University



Why, why, why…
…otherwise known as “Motivation”

- Most of what we do is high-level source transformations
  - We craft these ideas on papers, but we almost never implement them in the general case!
- Currently
  - I need a framework for
    - Testing different memory hierarchy models;
    - Generate different MMM versions;
  - Dan needs a framework for source to source transformation of C code in order to do application level checkpointing
  - Rohit needs a framework for applying complex transformations to pointer based codes (like ray tracing)
- A unified approach for Transformation and Translation
- Added value of having our own compiler test-bed

First feelings: huge project! Can we do it?

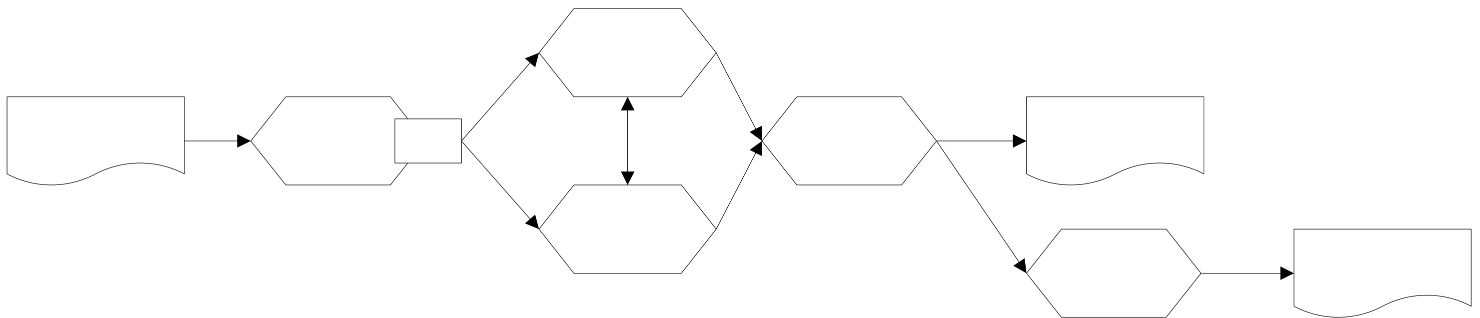
- What we want?
  - Easy to learn and use
  - Short & Comprehensible implementation
  - No proprietary tools for transformations
  - As open as possible
- How do we get it?
  - XML ASTs
  - Well structured code

- What we don't want:
  - Steep learning curve
    - SUIF, OpenC++
  - An API to learn…
  - A specific library and tools to use
  - Compilation speed compared to industrial compilers
    - Better said we are willing to sacrifice this in exchange for power!

- Expose SUIF-compatible interface to our XML AST structure
  - Will be able to run existing SUIF passes
- Maybe similar approach for OpenC++
- Maybe even ATLAS sort of thing…
- Make it easy for researchers to expose their favorite compiler-construction interface to our trees
- Facilitate interoperability with libraries like OMEGA
- This way we can have a clean back-end that we understand, while we will be able to leverage other’s work in the field

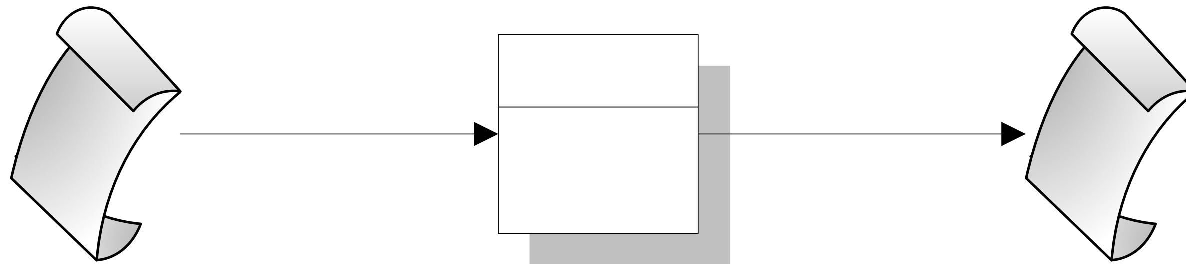
Example

- Some non-trivial Fortran code
- Calculate dependence relations (Omega)
- Transform imperfectly nested loop structure (Nawaaz)
- Perform several other optimizations (SUIF)
- Perform tiling / unrolling (our stuff)
- Spill out Fortran or C code



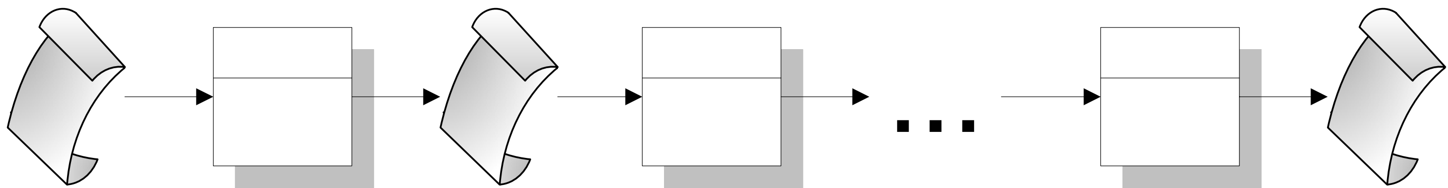
Step by step… 1

- An attractive goal:
  - Implement a general enough source to source transformation tool



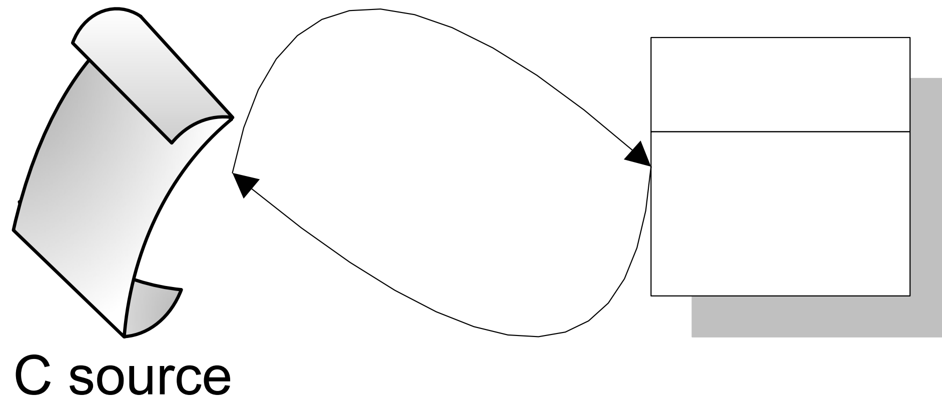
Step by step… 2

- Successively applying multiple transformations



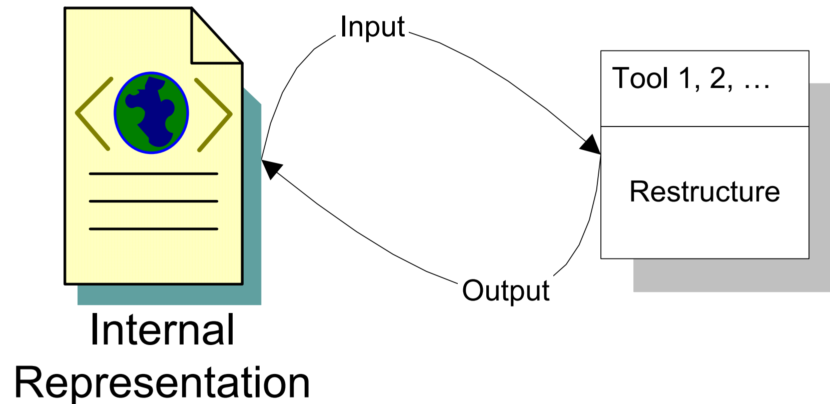
Step by step… 3

- Discovering the loop here
  - Think about source as an abstract type (instead as an instance of that type)



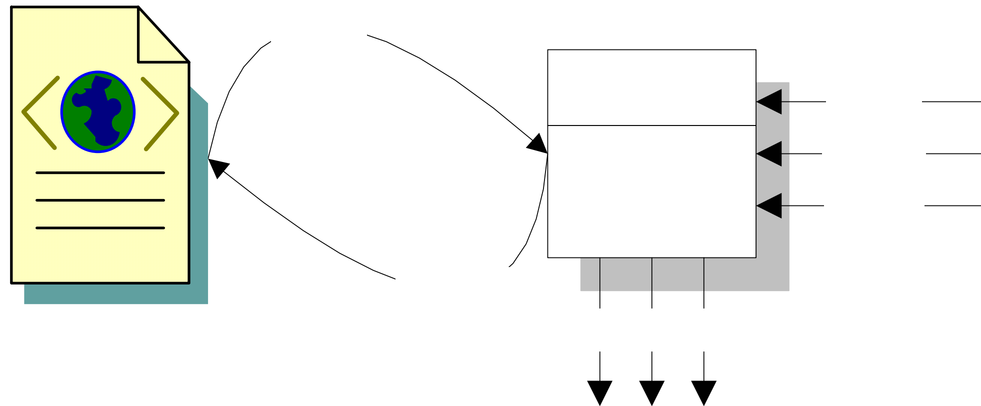
Step by step… 4

- Modifying source directly…
  - …is not the right answer!
  - It is easier to work with an abstract syntax tree
  - What tree?



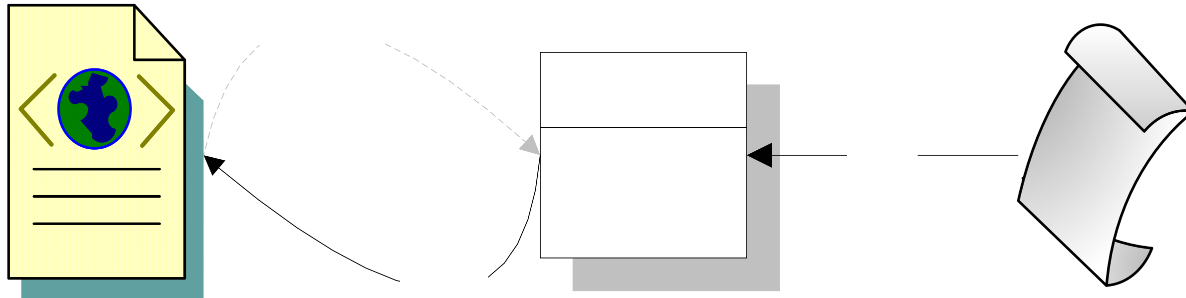
Step by step… 5

- Think global… the AST is not always enough!
  - We often need parameters…
  - We often need to produce other results…



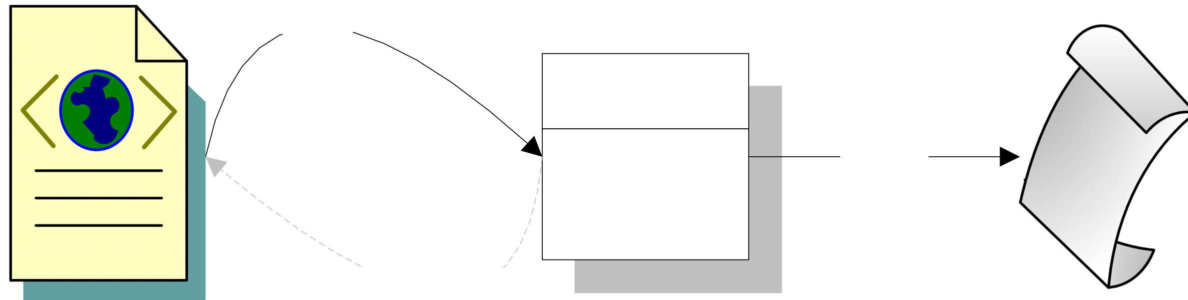
Step by step… 6

- How did we get an AST in first place?
- Nothing special…
  - Just another transform!



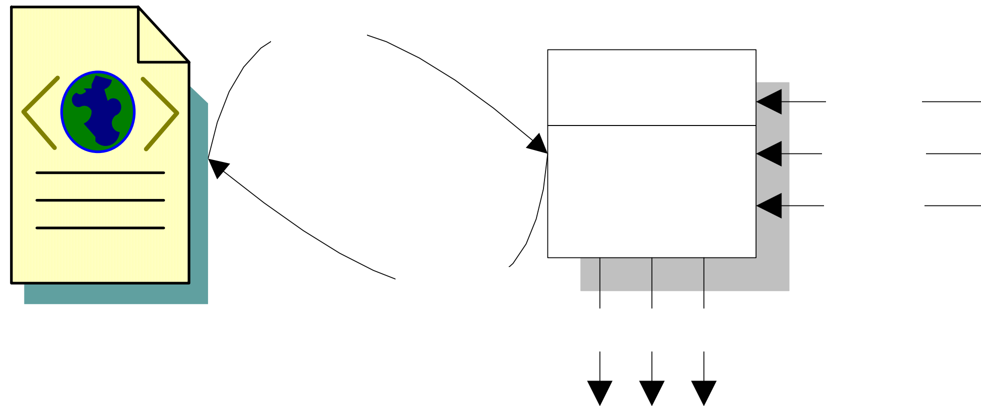
Step by step… 7

- What do we do with the final AST?
- Again nothing special…
  - Just another transform!



Step by step… 8

- The general view…
  - All other are special cases
- What does the AST contain?
- How is it represented?

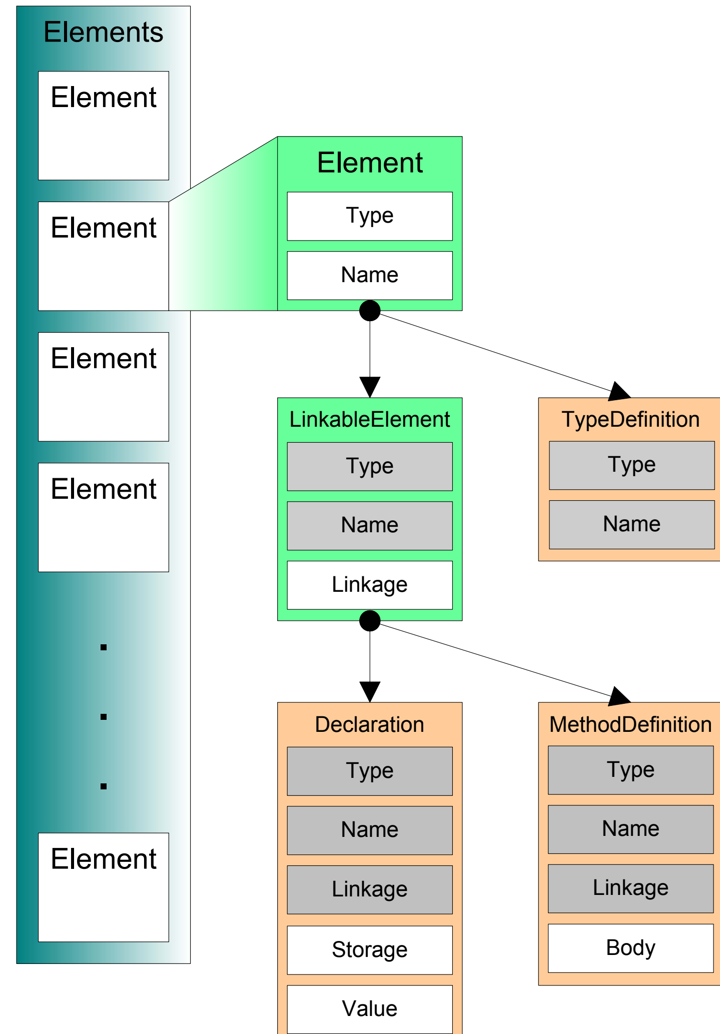


The Abstract Syntax Tree Notation

- Rectangles are Classes
  - Green are abstract
  - Orange can be instantiated
- Inner white rectangles are fields
- Names are only prefixes of original names
- Arrows mean inheritance

The Abstract Syntax Tree

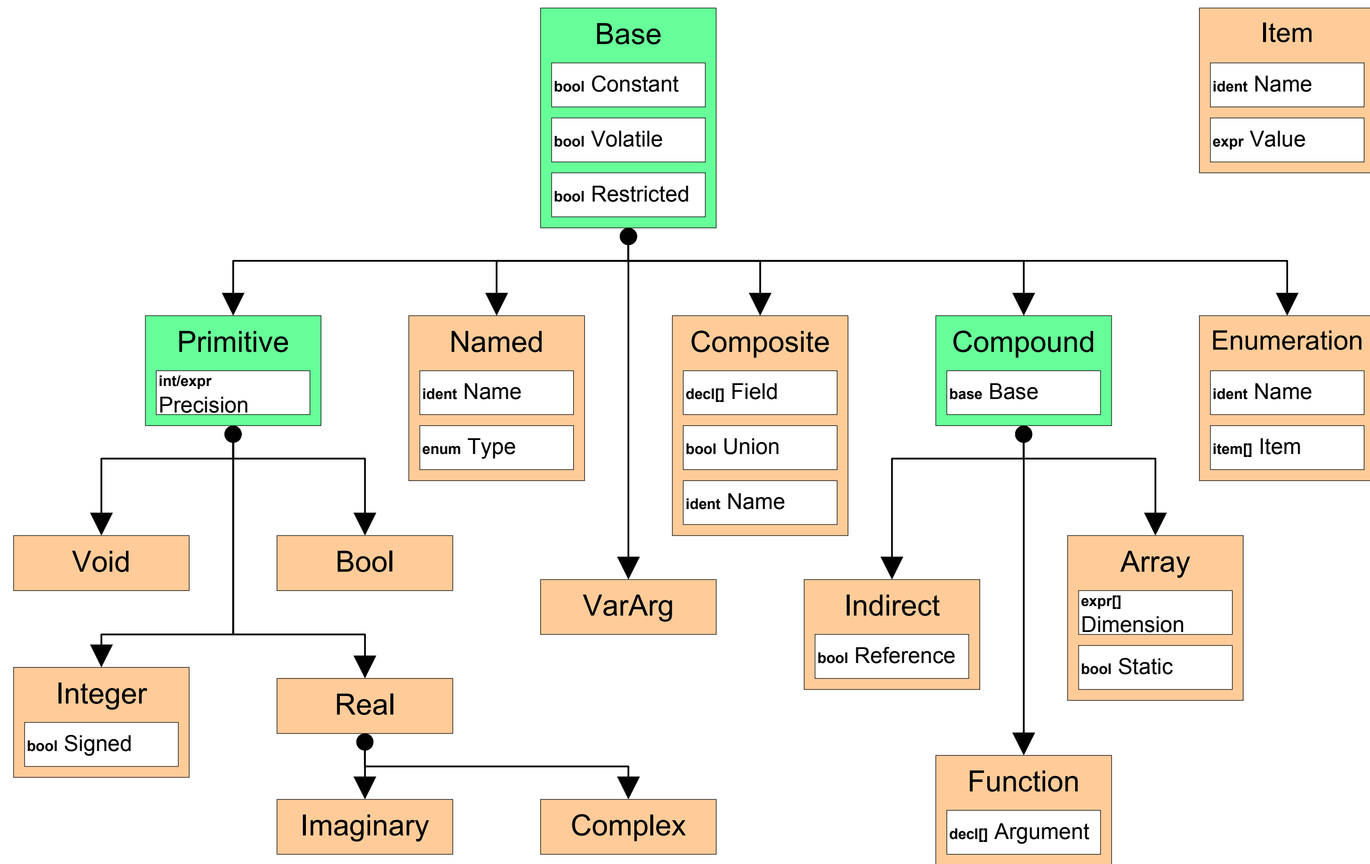
- Root: Elements
- Each element can be:
  - Declaration
  - TypeDefinition
  - MethodDefinition
- Abstract Classes
  - Element
  - LinkableElement
- Fields
  - Type
  - Name
  - Linkage & Storage
  - Value & Body



The Abstract Syntax Tree (cont.)

- Fields
  - Type: Type
    - discussed later
  - Name: Identifier
    - string
  - Linkage: Enumeration
    - External
    - Internal
    - None
  - Storage: Enumeration
    - Static
    - Register
    - Automatic
  - Value: Initializer
    - discussed later
  - Body: CompoundStatement
    - discussed later

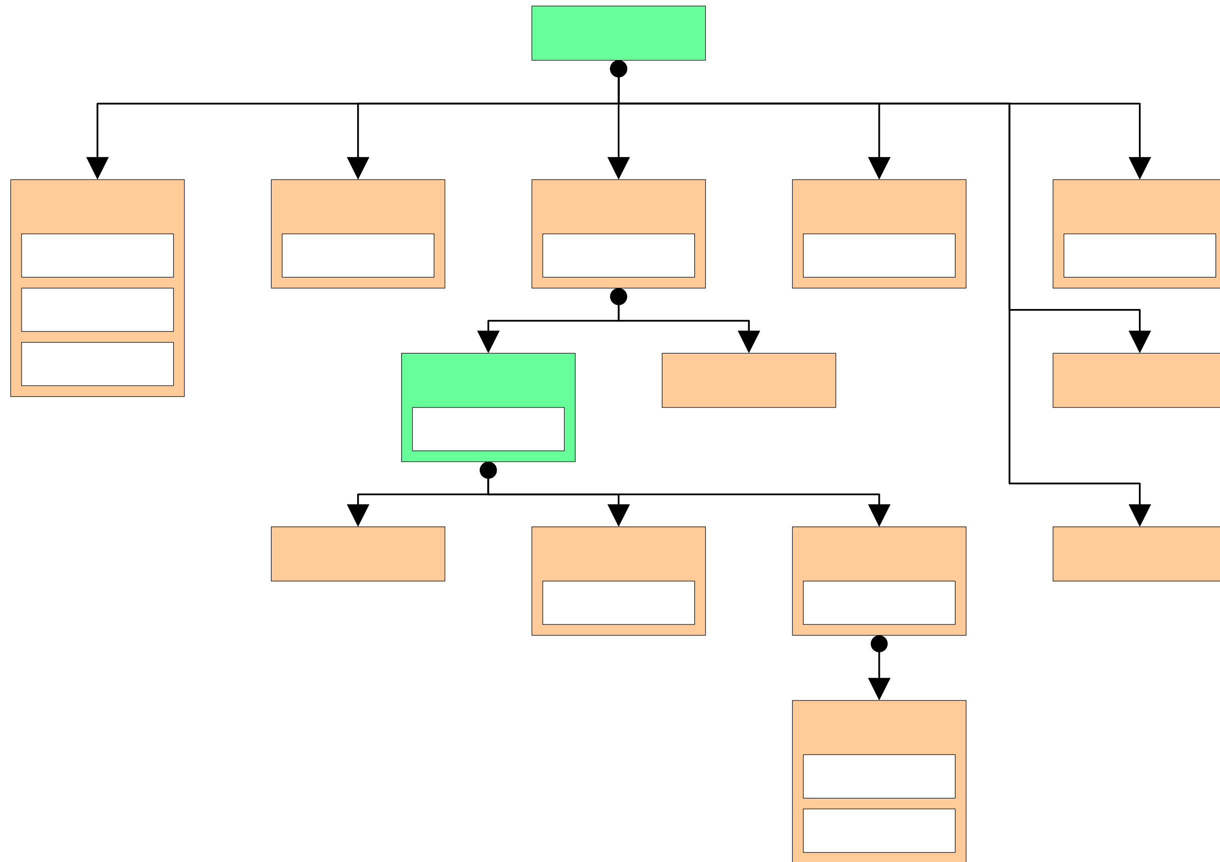
The Abstract Syntax Tree Types



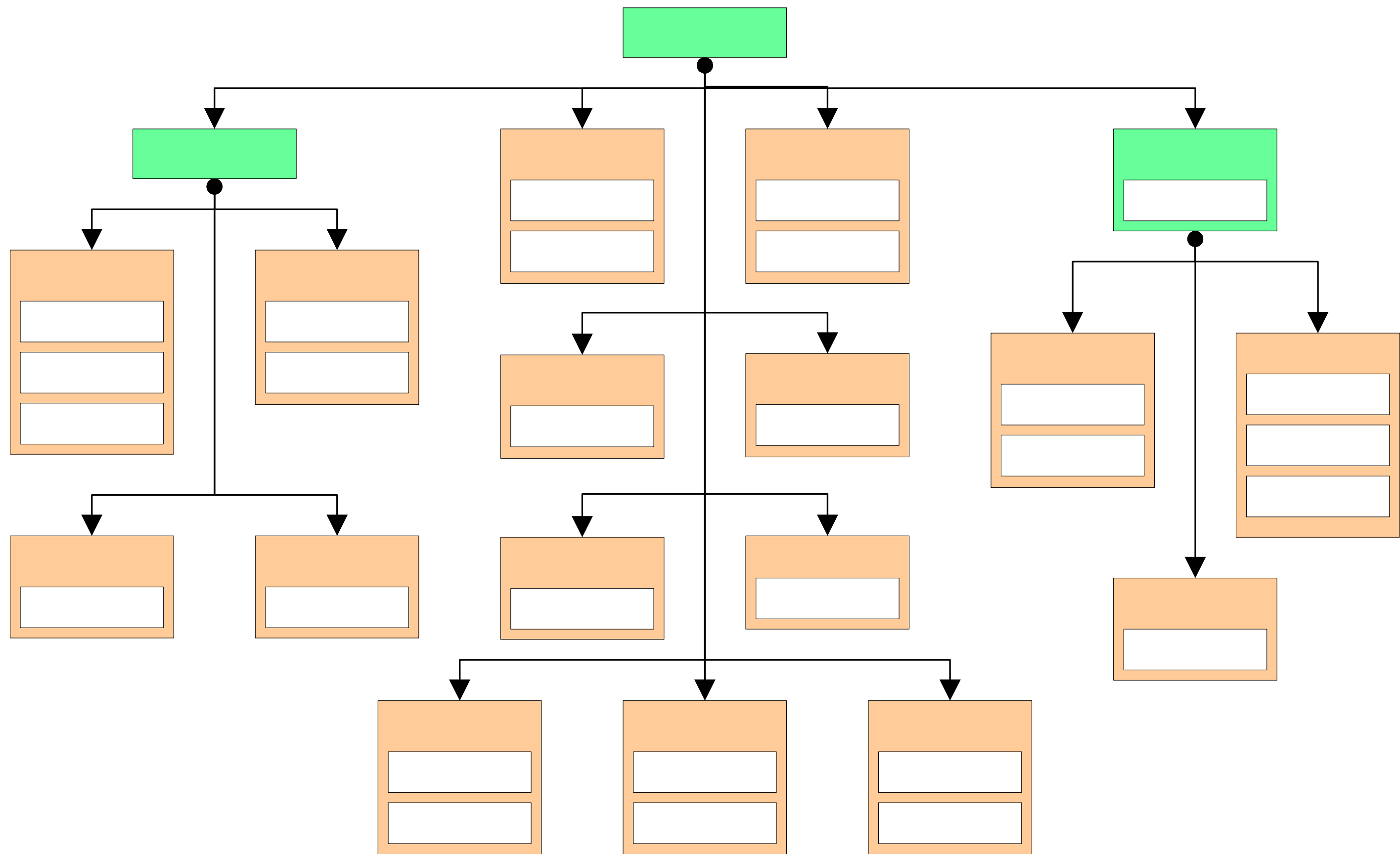
The Abstract Syntax Tree Types

- Named type
  - Enumeration
    - Type definition
    - Structure
    - Union
    - Enumeration (enum)

The Abstract Syntax Tree Statements

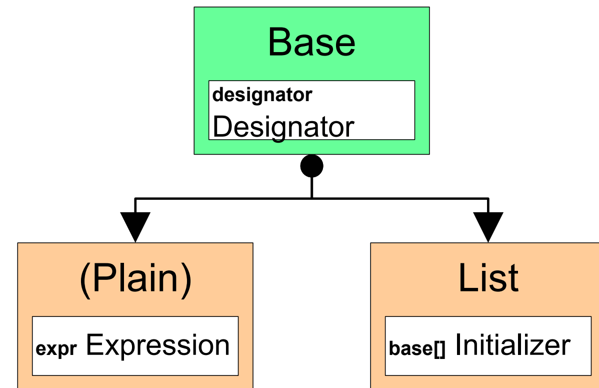


The Abstract Syntax Tree Expressions

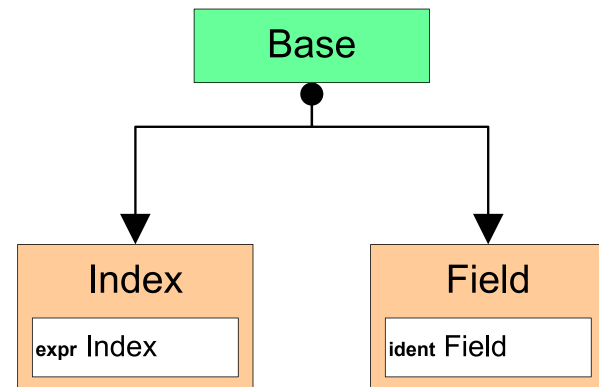


The Abstract Syntax Tree Initializers

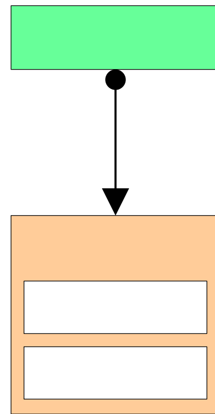
- Initializers

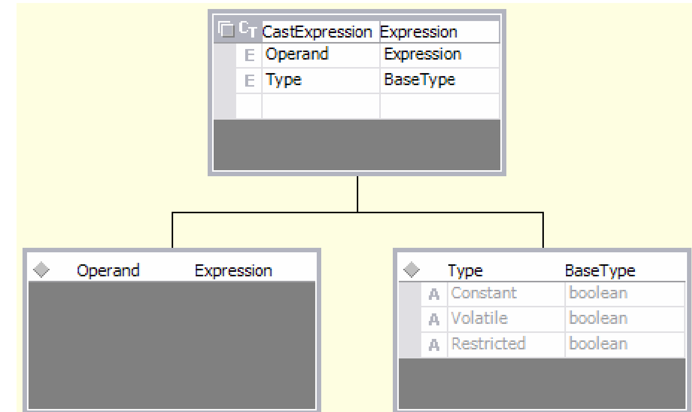


- Designators



The Abstract Syntax Tree Implementation





- Theory: Defined as XML Schema
- Practice:
  - Visually designed in VS.NET

The Abstract Syntax Tree Implementation (inside)

- How it looks from inside...

```xml
<xs:complexType name="CastExpression">
    <xs:complexContent>
        <xs:extension base="Expression">
          <xs:sequence>
                <xs:element name="Operand" type="Expression" />
                <xs:element name="Type" type="BaseType" />
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

- But you never write this!

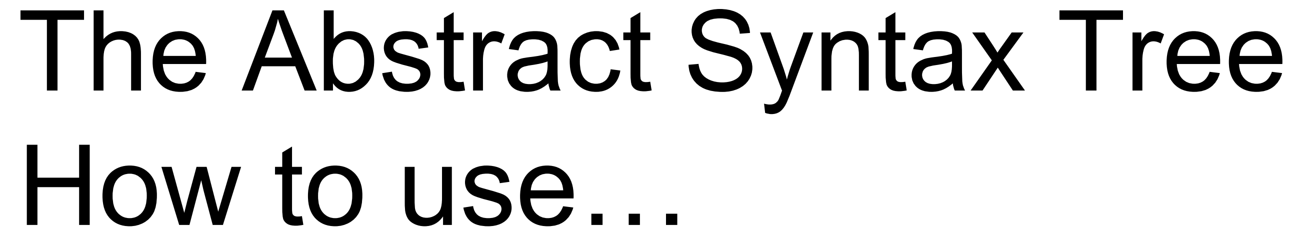
- How it looks from inside (converted)...

```
/// <remarks/>
[System.Xml.Serialization.XmlTypeAttribute(
    Namespace="http://bernoulli.cs.cornell.edu/ASTs.xsd")
]
public class CastExpression : Expression {

    /// <remarks/>
    public Expression Operand;

    /// <remarks/>
    public BaseType Type;
}
```

- You get this from running XSD on the XML Schema



- All these base types have single base class (Node)
  - Anotations
  - Unique ID / RefID
- You can persist it from/to XML…
- …but use it as it is a class library
- Available
  - To other languages
    - Through their special tools
  - To XML libraries and technologies
    - XPath
    - XSLT

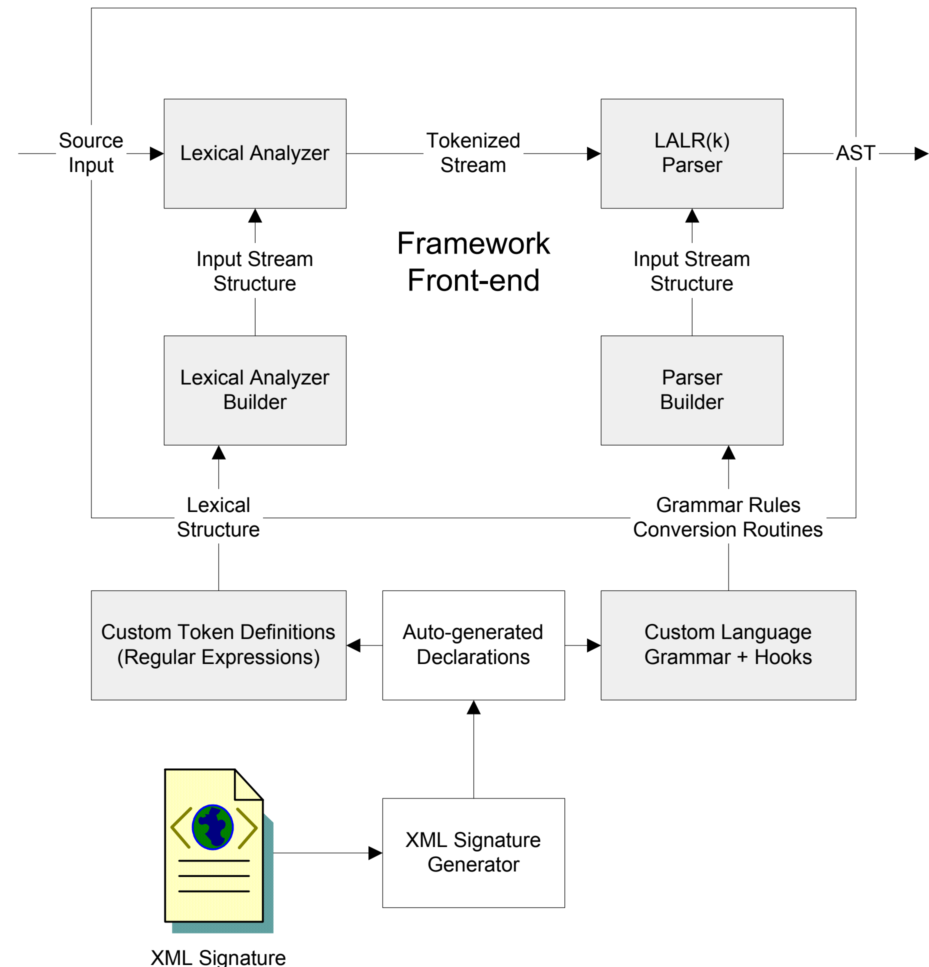
The Abstract Syntax Tree Tweaks

- Automatically generated
  - C# AST Class Library
  - Walk transformation
    - Virtual functions for all kinds of nodes
    - Example for Expression Statement

```
public virtual void VisitExpressionStatement(BC.Core.ASTs.ExpressionStatement _) {
    if (_ is CombinationStatement) {
        this.VisitCombinationStatement(((BC.Core.ASTs.CombinationStatement)(_)));
    }
    else {
        if (_ is ReturnStatement) {
            this.VisitReturnStatement(((BC.Core.ASTs.ReturnStatement)(_)));
        }
        else {
            System.Diagnostics.Debug.Assert(false, "invalid ExpressionStatement");
        }
    }
}
```

Some Internals
Parser detail

- Building a configuration
  - XML Signature
    - Token taxonomy
    - Keywords
    - Non-Terminals
  - Lexical analyzer
    - RegEx based
    - Source oriented
  - Syntactic analyzer
    - LALR(k) parsing ++
      - Recursion
    - Source oriented
  - Type-safe definitions
  - Clean implementation
    - Somewhat slow…
    - … but reliable!



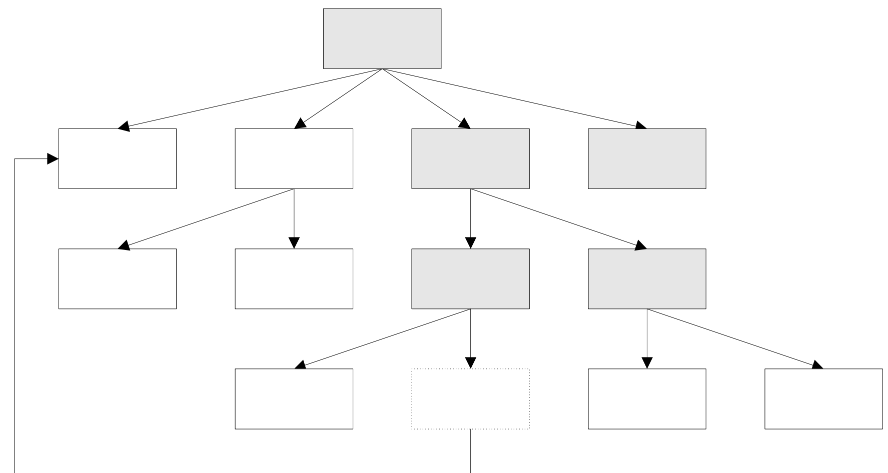
$$
\begin{aligned}
\mathbf{S} &\rightarrow \mathbf{X} \\
\mathbf{X} &\rightarrow \mathbf{X} + \mathbf{A} \\
&\mid\ \mathbf{A} \\
\mathbf{A} &\rightarrow \mathbf{A} * \mathbf{F} \\
&\mid\ \mathbf{F} \\
\mathbf{F} &\rightarrow (\ \mathbf{X}\ ) \\
&\mid\ \textbf{number} \\
&\mid\ \textbf{variable}
\end{aligned}
$$

XML Signature

```xml
<?xml version="1.0" encoding="utf-8" ?>
<tn:root xmlns:tn="http://bernoulli.cs.cornell.edu/Token.xsd" namespace="BC.X" class="X">
    <tn:roottoken identity="token" declaration="false">
        <tn:token identity="keyword" declaration="false" />
        <tn:token identity="asterisk" />
        <tn:token identity="primitive">
            <tn:token identity="number" />
            <tn:token identity="variable" />
        </tn:token>
        <tn:token identity="punctuator" declaration="false" >
            <tn:token identity="operator" declaration="false">
                <tn:token identity="add" name="O_add" />
                <tn:token identity="multiply" name="O_multiply"
                        bind="//tn:token[attribute::identity='asterisk']" />
            </tn:token>
            <tn:token identity="grouping" declaration="false">
                <tn:token identity="left" name="P_left" />
                <tn:token identity="right" name="P_right" />
            </tn:token>
        </tn:token>
    </tn:roottoken>
    <tn:keywords />
    <tn:nonterminals>
        <tn:nonterminal>S</tn:nonterminal>
        <tn:nonterminal>X</tn:nonterminal>
        <tn:nonterminal>A</tn:nonterminal>
        <tn:nonterminal>F</tn:nonterminal>
    </tn:nonterminals>
</tn:root>
```

Token Taxonomy

- Hierarchical structure
- Declarative power
- Virtual / Physical tokens
- …



XML Signature…
…translated

```
namespace BC.X {
        public class X {
                public static string[] keywords = new string[0];
                public static BC.Core.Grammars.Terminal T_asterisk = new BC.Core.Grammars.Terminal("_asterisk");
                public static BC.Core.Grammars.Terminal T_primitive = new BC.Core.Grammars.Terminal("_primitive");
                public static BC.Core.Grammars.Terminal T_number = new BC.Core.Grammars.Terminal("_primitive_number");
                public static BC.Core.Grammars.Terminal T_variable = new
                    BC.Core.Grammars.Terminal("_primitive_variable");
                public static BC.Core.Grammars.Terminal TO_add = new
                    BC.Core.Grammars.Terminal("_punctuator_operator_add");
                public static BC.Core.Grammars.Terminal TO_multiply = T_asterisk;
                public static BC.Core.Grammars.Terminal TP_left = new
                    BC.Core.Grammars.Terminal("_punctuator_grouping_left");
                public static BC.Core.Grammars.Terminal TP_right = new
                    BC.Core.Grammars.Terminal("_punctuator_grouping_right");
                public static BC.Core.Grammars.NonTerminal NS = new BC.Core.Grammars.NonTerminal("S");
                public static BC.Core.Grammars.NonTerminal NX = new BC.Core.Grammars.NonTerminal("X");
                public static BC.Core.Grammars.NonTerminal NA = new BC.Core.Grammars.NonTerminal("A");
                public static BC.Core.Grammars.NonTerminal NF = new BC.Core.Grammars.NonTerminal("F");
                private static System.Collections.Hashtable hKeywords = new System.Collections.Hashtable();
                public static BC.Core.Grammars.Terminal T_keyword(string name)
                {
                        if ((hKeywords.Contains(name) == false))
                                hKeywords[name] = new BC.Core.Grammars.Terminal(("_keyword_" + name));
                        return ((BC.Core.Grammars.Terminal)(hKeywords[name]));
                }
        }
}
```

Lexical analysis

- Source

```
sealed class ELexer: Lexer
{
	protected override void Configure ()
	{
		this[X.T_number] = @"[0-9]+";
		this[X.T_variable] = @"[a-z]+";
		this[X.TO_add] = @"\+";
		this[X.TO_multiply] = @"\*";
		this[X.TP_left] = @"\(";
		this[X.TP_right] = @"\)";
		this[WhiteSpace] = @"[ \t\v\r\n]+";
	}
}
```

- Generic form
  - Tokens $t_1, t_2, t_3, \ldots t_n$
  - Patterns $r_1, r_2, r_3, \ldots r_n$
  - Result:

$$(?<token>(?<t_1>r_1)| (?<t_2>r_2)\ldots (?<t_n>r_n))^*$$

- Final result

```
(?<token>
(?<_primitive_number>[0-9]+)|
(?<_primitive_variable>[a-z]+)|
(?<_asterisk>\*)|
(?<_whitespace>[ \t\v\r\n]+)|
(?<_punctuator_operator_add>\+)|
(?<_punctuator_grouping_right>\))|
(?<_punctuator_grouping_left>\()
)*
```

- Notes
  - This time fast
    - Natively compiled
  - Single string
  - .NET regular expressions grouping
  - _primitive_xxx

Syntactic Analysis

- Sample grammar definition
  - See how close it is to the initial format

```
sealed class EGrammar: Grammar
{
     . . .
     public EGrammar ()
     {
          this[X.NS]
                    = X.NX + new Reduction(_S);
          this[X.NX]
                    = X.NX + X.TO_add + X.NA + new Reduction(_X1)
                    | X.NA + new Reduction(_X2);
          this[X.NA]
                    = X.NA + X.TO_multiply + X.NF + new Reduction(_A1)
                    | X.NF + new Reduction(_A2);
          this[X.NF]
                    = X.T_primitive + new Reduction(_F1)
                    | X.TP_left + X.NX + X.TP_right + new Reduction(_F2);
     }
}
```

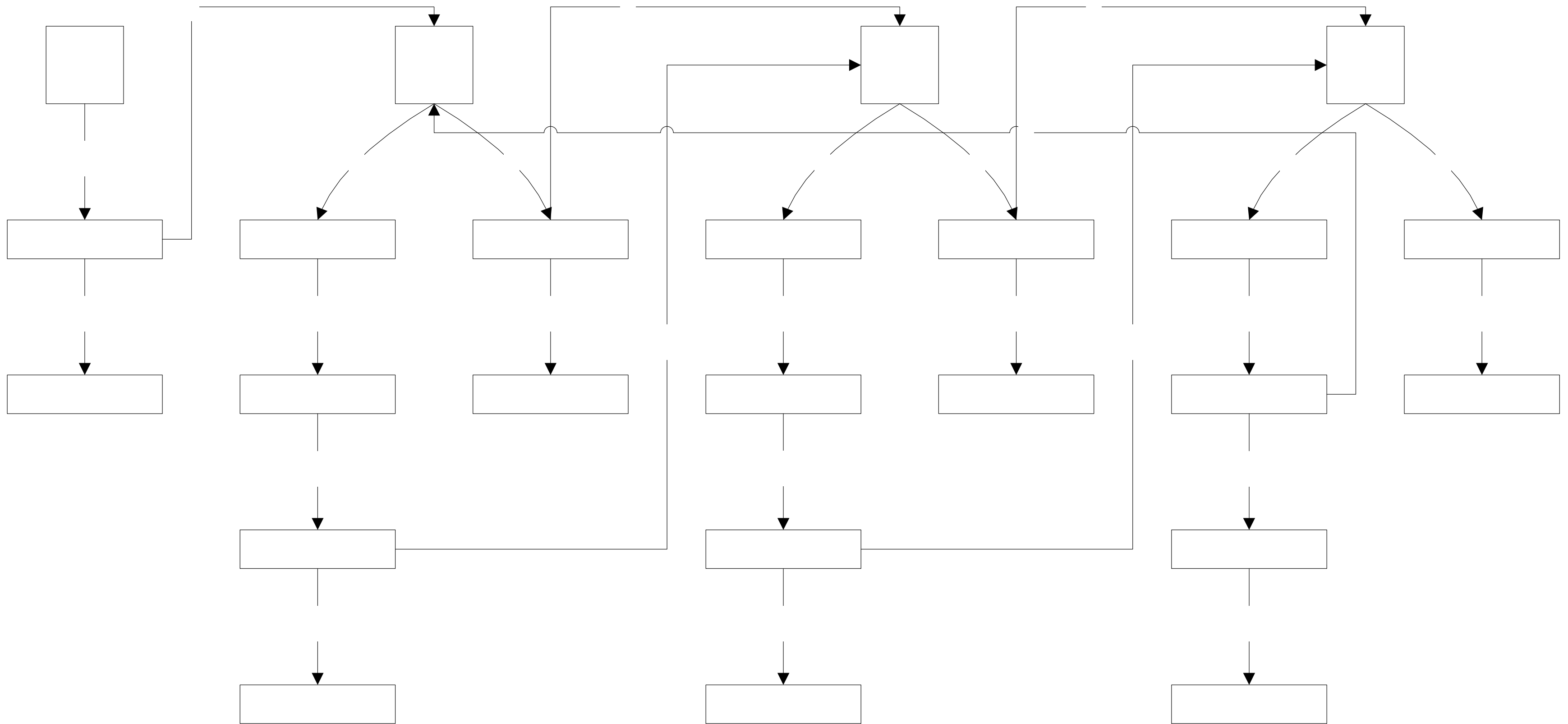
Syntactic analysis terminology

- *Symbol* is a *Terminal* or *Non-Terminal* (e.g. anything like X.Txxx and X.Nxxx);
- *Sequence* is several symbols concatenated with ‘+’ (e.g. X.NX + X.TO_add + X.NA);
- *Alternative* is a *Sequence* + *Reduction*
- *Reduction* is a object-member function closure, that gets called when the parser performs a reduction using the *Production* whose *Alternative* has the *Reduction* in question. Using .NET terminology, this is a delegate;
- *Production* is a *Non-Terminal* → *Alternative*;
- *Alternatives* is a list of *Alternative* elements concatenated with ‘|’;
- *Rule* is a *Non-Terminal* → *Alternatives*;
- *Grammar* is a set of *Rule* elements.

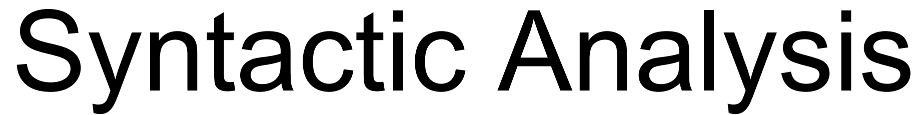
Syntactic Analysis Parsing Phase 1

- LR(0) NFA construction
  - Nodes
    - *Stations*, labeled with something like “• *Non-Terminal*” (e.g. “• S”). We create one of these for each non-terminal in the grammar;
    - *Items*, labeled with one of the productions in the grammar with a • somewhere in the middle of the RHS (e.g. “$X \rightarrow X \bullet + A$”). The intuition behind the • is that we want to perform a reduction using this production and we have reached that far in recognizing the RHS.
  - Edges
    - ε-edges from each station • N to all items in the form $N \rightarrow \bullet \ldots$ (i.e. dot in the beginning);
    - ε-edges from all items in the form $N \rightarrow \ldots \bullet N2 \ldots$ to the station • N2;
    - edge labeled with σ from $N \rightarrow \ldots \bullet \sigma \ldots$ to $N \rightarrow \ldots \sigma \bullet \ldots$;

Syntactic Analysis Parsing Phase 1 (picture)



- $First_k(N)$ construction
  - Non-terminal look-ahead sets
  - Inductive definition
    - $F_0(T) = \{<T>\}$
    - $F_i(N) = F_{i-1}(N)$ Union $\{ F_{i-1}(Y_1) \oplus_k F_{i-1}(Y_2) \oplus_k \ldots \oplus_k F_{i-1}(Y_m) \}$ for all $N \rightarrow Y_1 Y_2 \ldots Y_m$
    - $X \oplus_k Y$ = Union { xy truncated to the first k symbols } for all x in X and y in Y
    - $F_0(T) \supseteq \ldots \supseteq F_i(N) \supseteq \ldots \supseteq F_l(N) = F_{l+1}(N)$
    - $First_k(N) = F_l(N)$



Syntactic Analysis Parsing Phase 3, 4, 5, …

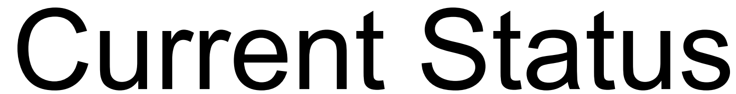
- Look-ahead set propagation
- DFA construction
  - Normal algorithm from discrete math
- Actual LALR(k) parsing
  - Described in detail in the books

The interesting part: Custom XML Transformations

- Tools
  - XSLT
  - XML, XSD, XPath libraries
- Transformations vs. Translations
  - A transformation converts XML ASTs preserving the schema
  - A translation converts XML ASTs from one schema to another (IR)

Some Internals
Renderer detail

- Not much to say, but…
  - We need to use the final XML tree
  - No current compilers understand XML
- Translations
  - To initial source code;
  - To some intermediate representation;
  - To assembly code;
  - To binary;



Current Status

- Working on C (ISO/IEC 9899:1999(E))
  - Completed lexer
  - Completed parser
  - Completed inter-language schema
  - Parses too much C
    - Checking pass is the solution, but is tedious
- Need you to tweak Polaris to output the XML ASTs we need…
- Need applications…but first:
  - MMM for the tomorrow part of the PLDI paper;
  - My memory performance models;
  - Nawaaz: the thesis;
  - Chatterjee: counting of cache misses (OMEGA);
  - Ghosh: Cache miss equations;
  - Dan’s Application-level Check-pointing;

- Lex & Yacc
  http://dinosaur.compilertools.net/
- XML
  http://www.w3.org/XML/
- XSD
  http://www.w3.org/XML/Schema
- XPath
  http://www.w3.org/TR/xpath
- Microsoft .NET framework
  http://www.microsoft.com/net/
- SUIF
  http://suif.stanford.edu/
- OpenC++
  http://www.csg.is.titech.ac.jp/~chiba/openc++.html
- Programming Language C
  http://webstore.ansi.org/ansidocstore/product.asp?sku=ANSI%2FISO%2FIEC+9899%2D1999
  (we bought this, so if you want it, let me know!)
- C# Language Specification
  http://msdn.microsoft.com/vstudio/techinfo/articles/upgrade/Csharpdownload.asp