XSLT EARLEY

First Steps to a Declarative Parser Generator

TOMOS HILLMAN

tom@eXpertML.com

XSLT Earley

- What am I talking about?
- Why would I do this?
- How did I do it?

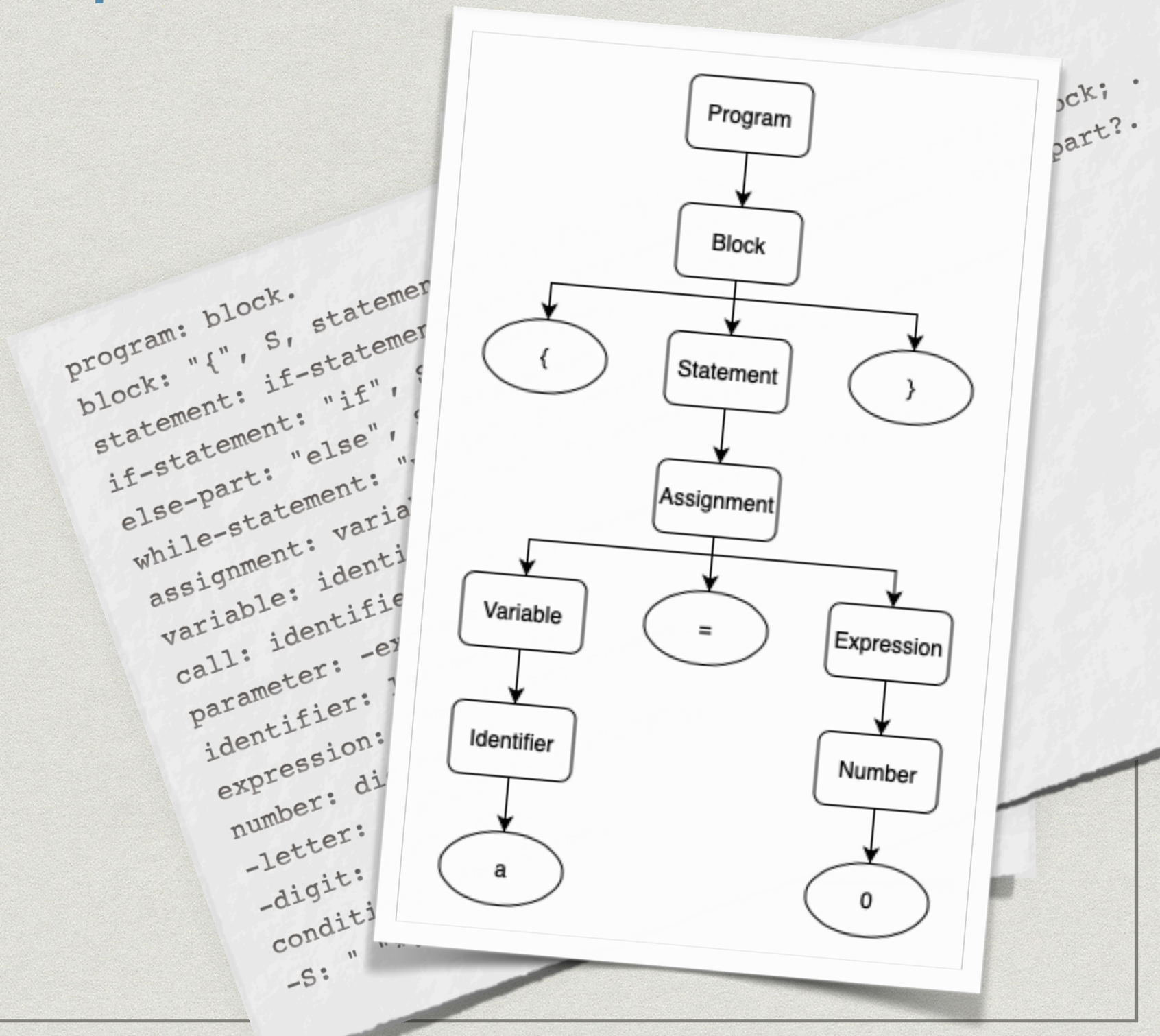
WHAT IS A PARSER?

WHAT IS ~~A PARSER~~ XSLT?

WHAT IS A PARSER?

What is a parser?

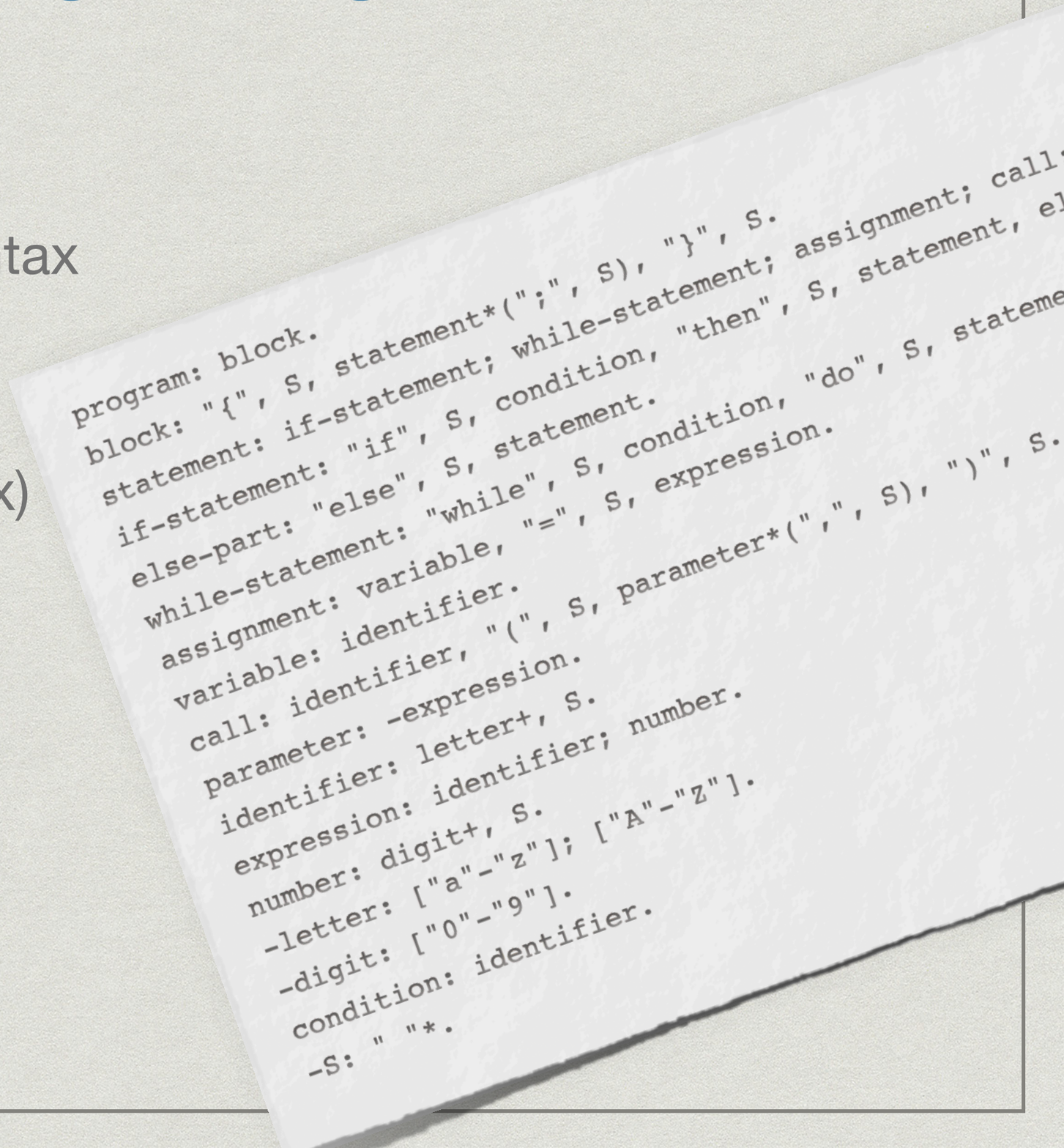
- Takes a string
- And a grammar
- Returns a tree



WHAT IS A GRAMMAR?

What is a Grammar?

- Defines a language's syntax
- Symbols
  - Terminal (strings, regex)
  - Nonterminals
- Repetition
- Optionality



WHY IS THIS USEFUL?

Invisible XML

- Invented/discovered by Steven Pemberton in 2013
- All data is an abstraction
- So expressions/representations can be thought of as equivalent
- If it can be parsed with a grammar, we can treat it like XML

WHY XSLT?

Why XSLT?

- May not be the most efficient?
- Not much use if you aren’t already using XSLT...

Why not XSLT?

- Pure XSLT available anywhere
- Embedded in the XML Stack
  - Schematron
  - XSpec
  - etc

Isn’t there an XSLT parser already?

- REx Parser Generator by Gunther Rademacher
- https://www.bottlecaps.de/rex/
- Very useful 90% of the time!
- LL1 parser - doesn’t support all grammars
- not extensible

EXTENSIBLE PARSERS?

Extensible Parsers

- REx parsers aren't supposed to be read by humans
- No scope for imports, next-match etc
- Can't change the input as you parse
  - cf DTD parsing with entities

CAN IT BE DONE?

TURNS OUT YES!

But let's talk XSLT at the end

Parser vs parser generator

- Lookahead parsers like REx can only process one particular grammar - needs a parser generator
- The XSLT Parser works by transforming the grammar as an input - no need for a parser generator!
- Sorry about the paper title...

Parsing other grammars

- We require a grammar expressed as XML - Invisible XML as XML
- But we have a tool for expressing non-XML grammars - let’s bootstrap!
- We can use ANY other grammar language which can be described using Invisible XML

Ambiguous Parsing

- There may be more than one valid way to parse a string
  - LL1 parsers can’t do this!
  - Earley parsers can - return the first, or return them all

Ambiguous Parsing II

- Since we can already return multiple results...
- Why not try parsing multiple grammars?
- language detection!

SHALL WE TALK ABOUT CODE?

Time permitting!

Earley Algorithm (briefly!)

Some useful terms:

- **State** tells you what you have left to parse
- **Rules** map one **symbol** to a sequence of **terminal** and **nonterminal** symbols

Earley Algorithm (briefly!)

Starting with the first rule of the grammar, build a table of “Earley Items” storing:

- The current rule
- The position in the current rule
- The current state
- The state when the rule evaluation began

Earley Algorithm (briefly!)

Once we have an item in the table that

- completes a rule that
- starts in the initial state
- (and ends in the end state)

we have a valid (complete) parse.

EARLEY TREES

Earley Trees

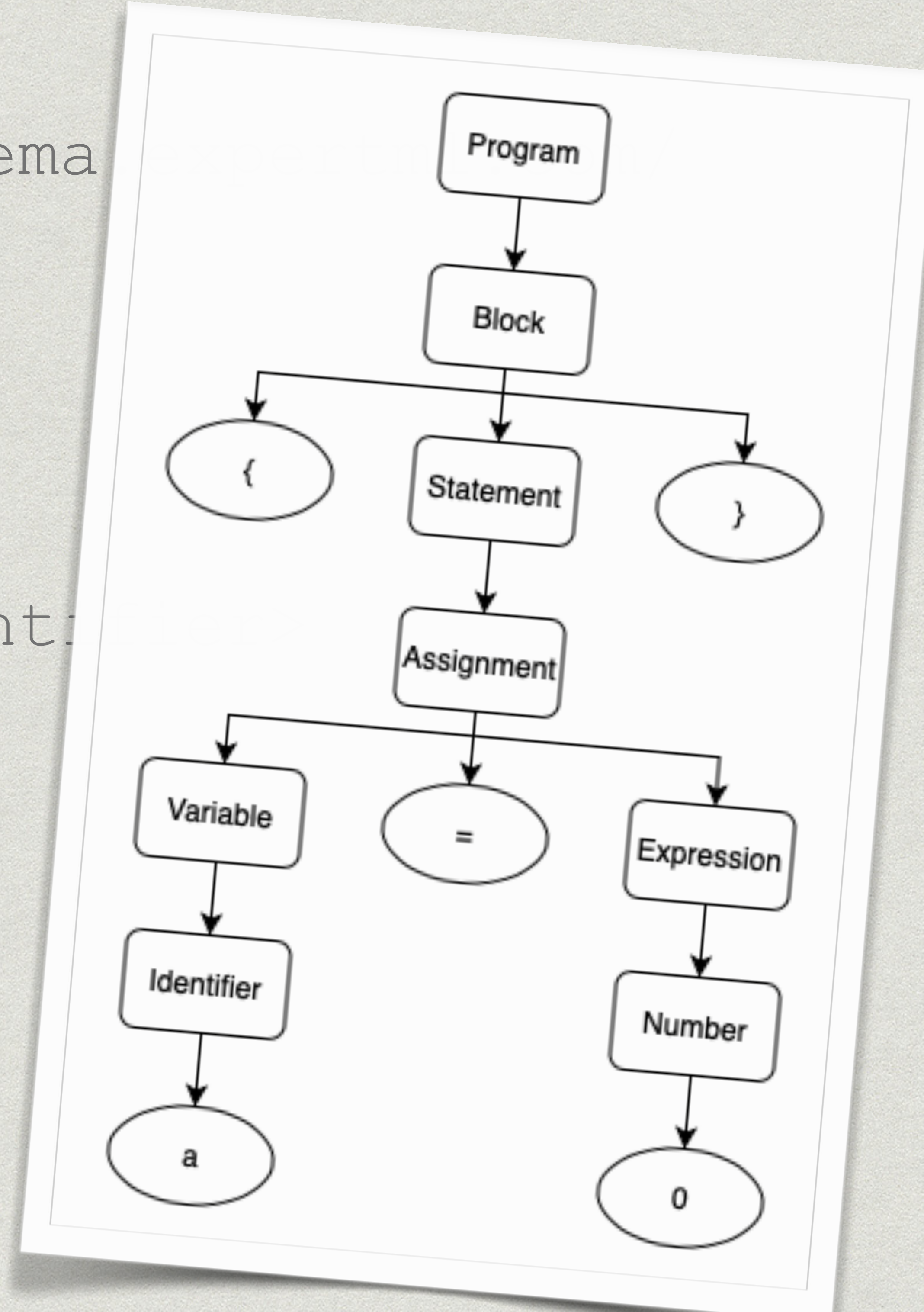
- **Rules** are stored as containing elements; children are one or more alternatives
- Position in the rule is position in the tree
- **State** is stored as a sequence of strings as a parameter
- Infinite recursion is avoided by holding a map of **visited** rules in particular states
- **Terminals** are held as ‘leaf’ nodes
- **Visited Nonterminals** are held as ‘leaf’ node ‘links’

Pruning the Earley Tree

- Exclude elements that fail or are empty
- Copy/expand nonterminal references
- Choose from alternatives & remove unnecessary structural elements

```
<program>
  <block xmlns:e="http://schema
EarleyParser">
                    </identi
                    mber>
  </block>
</program>
```

{a=0}



CONTROLLING PROCESS ORDER

Controlling Process Order

- The **visited** and **state** information needs to be passed:
  - from sibling to following sibling
  - from final sibling to parent
  - NOT from parents to children
- We need to override the default processing order (preferably preserving the pull processing idea)

Controlling Process Order

- Use named template instead of applying templates
- Process children as a variable before forming the parent
- Or calculate parent values separately (custom functions & modes)

OPTIONALITY & REPITITION

Rewriting repeat0

```
<xsl:variable name="GID" select="(@gid, generate-
id(.))[1]"/>
<xsl:variable name="equivalent" as="element(alts)">
  <alts gid="{$GID}">
    <alt>
      <empty/>
    </alt>
    <alt>
      <xsl:sequence
select="(child::*[not(self::sep)], sep)"/>
      <xsl:copy>
        <xsl:attribute name="gid" select="$GID"/>
        <xsl:copy-of select="@*, node()"/>
      </xsl:copy>
    </alt>
  </alts>
</xsl:variable>
```

END OF TALK

Thanks for listening!