Abstract LR-parsing

Kyung-Goo Doh[1], Hyunha Kim[1], David A. Schmidt[2]

[1] Hanyang University, Ansan, South Korea
[2] Kansas State University, Manhattan, Kansas, USA

Abstract. We combine LR(k)-parsing technology and data-flow analysis to analyze, in advance of execution, the documents generated dynamically by a program. Based on the document language's context-free reference grammar and the program's control structure, formatted as a set of flow equations, the analysis *predicts* how the documents will be generated and simultaneously *parses* the predicted documents. Recursions in the flow equations cause the analysis to emit a set of residual equations that are solved by least-fixed point calculation in the domain of *abstract (folded) LR-parse stacks*.

Since the technique accommodates LR(k) grammars, it can also handle string-update operations in the programs by translating the updates into finite-state transducers, whose controllers are composed with the LR(k)-parser controller.

1 Motivation

Scripting languages like PHP, Javascript, Perl, and Python use strings as a "universal data structure" to communicate values, commands, and programs. For example, one might write a PHP script that assembles within a string variable an SQL query or an HTML page or an XML document.

Typically, the well-formedness of the assembled string is verified when the string is supplied as input to its intended processor (database, web browser, or interpreter), and an incorrectly assembled string might cause processor failure. Worse still, a malicious user might deliberately supply misleading input that generates a document that attempts a cross-site-scripting or injection attack.

As a first step towards preventing failures and attacks, the well-formedness of a dynamically generated, "grammatically structured" string (document) should be checked with respect to the document's context-free *reference grammar* (for SQL or HTML or XML) before the document is supplied to its processor. Better still, the document generator program *itself* should be analyzed to validate that all its generated documents are well formed with respect to the reference grammar, like an application program is type checked in advance of execution.

2 Motivating example

Say that a script must generate an output string that conforms to this grammar,

$$S \rightarrow \mathsf{a} \mid [\, S \,]$$

where S is the only nonterminal. (HTML, XML, and SQL are such bracket languages.) The grammar is LR(0), but it can be difficult to enforce even for simple programs, like the one in Figure 1, left column. Perhaps we require this program to print only well-formed S-phrases — the occurrence of
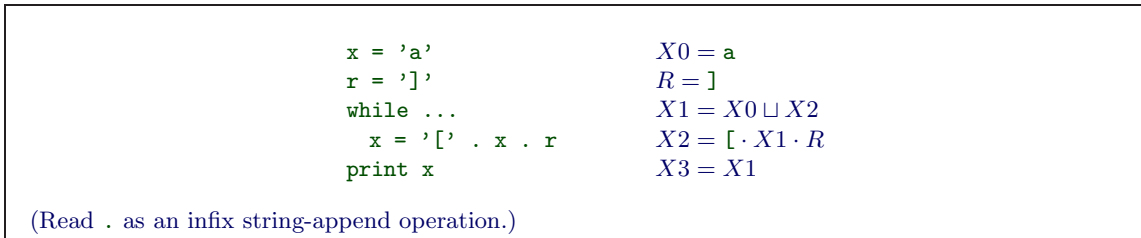
```
x = 'a'              X0 = a
r = ']'              R = ]
while ...            X1 = X0 ⊔ X2
  x = '[' . x . r    X2 = [ · X1 · R
print x              X3 = X1
```

(Read . as an infix string-append operation.)

Fig. 1. Sample program and its data-flow equations

x at "`print x`" is a "hot spot" and we must analyze x's possible values.

1. A *type checking* analysis assigns types (reference-grammar nonterminals) to the program's variables. The occurrences of x can indeed be data-typed as S, but r has no data type that corresponds to a nonterminal.
2. A *regular expression* analysis (Christensen [2], Minamide [8], Wasserman [10]) solves the flow equations in Figure 1 in the domain of regular expressions, determining that the hot spot's ($X3$'s) values conform to the regular expression, $[^* \cdot \mathsf{a} \cdot]^*$, but this does not validate the assertion.
3. A *grammar-based analysis* (Thiemann [9]) treats the flow equations as a set of grammar rules. A language-inclusion check base tries to prove that all $X3$-generated strings are S-generable.

Our approach solves the flow equations in the domain of *parse stacks* — $X3$'s meaning is the *set of LR-parses* of the strings that might be denoted by x. The technique "unfolds" the strings denoted by $X3$ at the same time that it executes the LR(k) parser. This is more precise than the techniques listed above.
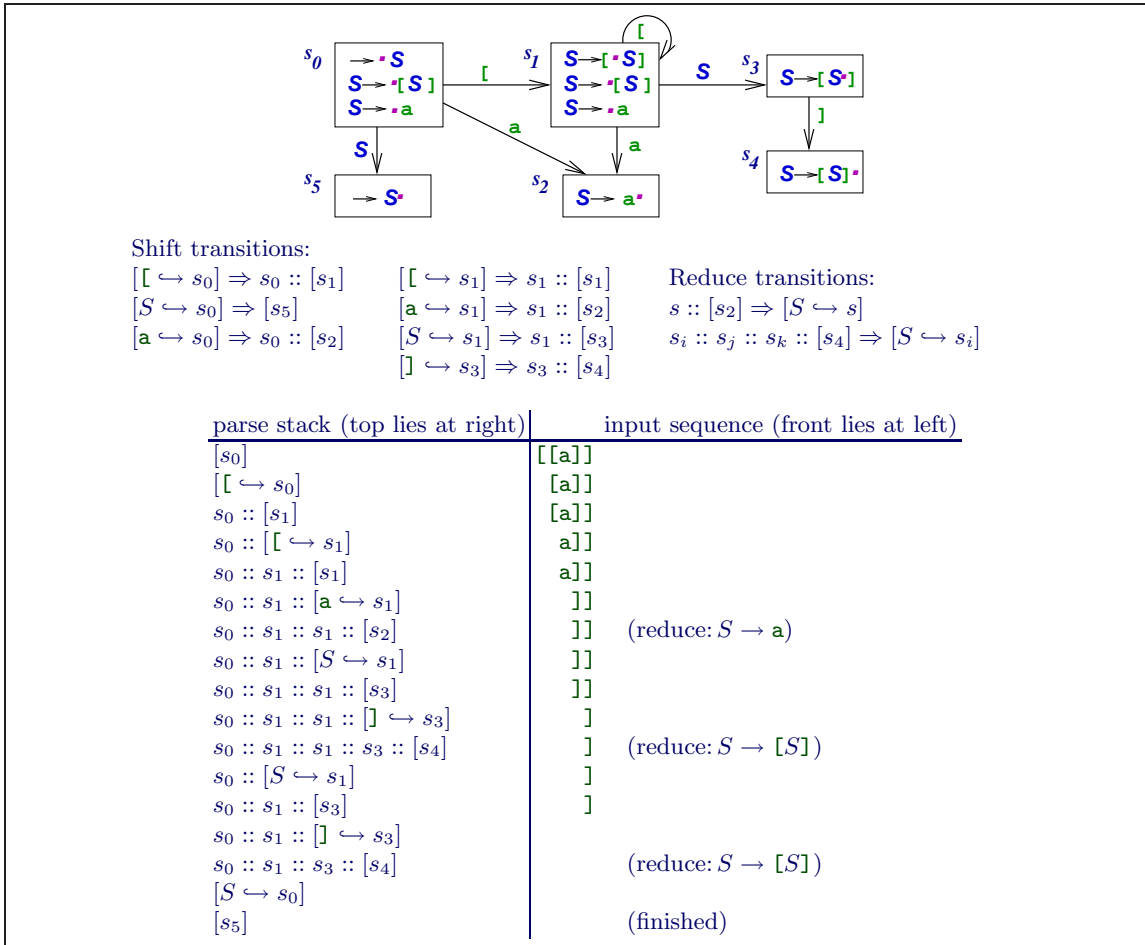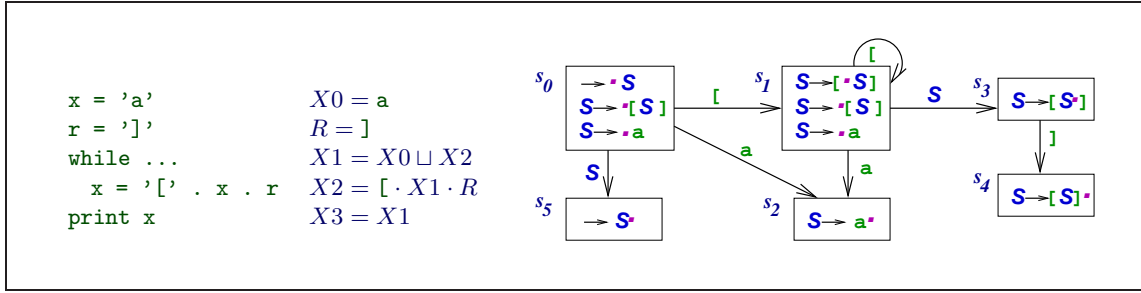
Shift transitions:

$[[\hookrightarrow s_0] \Rightarrow s_0 :: [s_1]$ $[[\hookrightarrow s_1] \Rightarrow s_1 :: [s_1]$ Reduce transitions:

$[S \hookrightarrow s_0] \Rightarrow [s_5]$ $[a \hookrightarrow s_1] \Rightarrow s_1 :: [s_2]$ $s :: [s_2] \Rightarrow [S \hookrightarrow s]$

$[a \hookrightarrow s_0] \Rightarrow s_0 :: [s_2]$ $[S \hookrightarrow s_1] \Rightarrow s_1 :: [s_3]$ $s_i :: s_j :: s_k :: [s_4] \Rightarrow [S \hookrightarrow s_i]$

$[] \hookrightarrow s_3] \Rightarrow s_3 :: [s_4]$

parse stack (top lies at right)	input sequence (front lies at left)
$[s_0]$	[[a]]
$[[\hookrightarrow s_0]$	[a]]
$s_0 :: [s_1]$	[a]]
$s_0 :: [[\hookrightarrow s_1]$	a]]
$s_0 :: s_1 :: [s_1]$	a]]
$s_0 :: s_1 :: [a \hookrightarrow s_1]$	]]
$s_0 :: s_1 :: s_1 :: [s_2]$	]] (reduce: $S \to$ a)
$s_0 :: s_1 :: [S \hookrightarrow s_1]$	]]
$s_0 :: s_1 :: s_1 :: [s_3]$	]]
$s_0 :: s_1 :: s_1 :: [] \hookrightarrow s_3]$	]
$s_0 :: s_1 :: s_1 :: s_3 :: [s_4]$	] (reduce: $S \to [S]$)
$s_0 :: [S \hookrightarrow s_1]$	]
$s_0 :: s_1 :: [s_3]$	]
$s_0 :: s_1 :: [] \hookrightarrow s_3]$	
$s_0 :: s_1 :: s_3 :: [s_4]$	(reduce: $S \to [S]$)
$[S \hookrightarrow s_0]$	
$[s_5]$	(finished)

Fig. 2. Parse controller for $S \to [S] \,|\, $ a   and an example parse of [[a]]

```
x = 'a'            X0 = a
r = ']'            R = ]
while ...          X1 = X0 ⊔ X2
  x = '[' . x . r  X2 = [ · X1 · R
print x            X3 = X1
```

s_0: → ·S ; S → ·[S] ; S → ·a ; S

s_1: S → [·S] ; S → ·[S] ; S → ·a

s_3: S → [S·]

s_4: S → [S]·

s_2: S → a·

s_5: → S·

To analyze the hot spot at $X3$, we must $LRparse(X3, s_0)$, which we portray as a function call, $X3[s_0]$ — *this denotes a set of LR-parse stacks*. The flow equation, $X3 = X1$, generates this call:

$$X3[s_0] = X1[s_0]$$

which demands a parse of the strings generated at $X1$ from state s_0:

$$X1[s_0] = X0[s_0] \cup X2[s_0]$$

The union of the parses of strings at $X0$ and $X2$ from s_0 must be computed. (This computes *sets of parse stacks*. In this example, all the sets are singletons.) Consider $X0[s_0]$:

$$X0[s_0] = [\mathtt{a} \hookrightarrow s_0] \Rightarrow s_0 :: [s_2] \Rightarrow [S \hookrightarrow s_0] \Rightarrow [s_5].$$

That is, a parse of '$\mathtt{a}$' from s_0 generates the one-element stack, s_5 (actually, $\{[s_5]\}$) — all strings denoted by $X0$ are S-phrases. Next,

$$\begin{aligned}
X2[s_0] &= ([\cdot X1 \cdot R)[s_0] = [[\hookrightarrow s_0] \oplus (X1 \cdot R) \\
&\Rightarrow (s_0 :: [s_1]) \oplus (X1 \cdot R) \\
&= s_0 :: (X1 \cdot R)[s_1] = s_0 :: (X1[s_1] \oplus R)
\end{aligned}$$

For parse stack, st, and function, E, define $st \oplus E = tail(st) :: E[head(st)]$. That is, stack st's top state feeds to E. Next, $X1[s_1] = X0[s_1] \cup X2[s_1]$ computes to $s_1 :: [s_3]$ (explained below!), so

$$\begin{aligned}
X2[s_0] &= s_0 :: (X1[s_1] \oplus R) = (s_0 :: s_1 :: [s_3]) \oplus R = s_0 :: s_1 :: R[s_3] = s_0 :: s_1 :: [] \hookrightarrow s_3] \\
&\Rightarrow s_0 :: s_1 :: s_3 :: [s_4] \Rightarrow [S \hookrightarrow s_0] \Rightarrow [s_5]
\end{aligned}$$

That is, $X2[s_0]$ built the stack, $s_0 :: s_1 :: s_3 :: s_4$, denoting a parse of $[S]$, which reduced to S, giving s_5.

```
x = 'a'                   X0 = a
r = ']'                   R = ]
while ...                 X1 = X0 ⊔ X2
  x = '[' . x . r         X2 = [ · X1 · R
print x                   X3 = X1
```

Here is the list of generated function calls. You can pretend they are the *residual functions* generated from a *partial evaluation* [**?**] of the initial call, $X3[s_0]$:

$$X3[s_0] = X1[s_0]$$
$$X1[s_0] = X0[s_0] \cup X2[s_0]$$
$$X0[s_0] = [s_5]$$
$$X2[s_0] = s_0 :: (X1[s_1] \oplus R)$$
$$X1[s_1] = X0[s_1] \cup X2[s_1]$$
$$X0[s_1] = s_1 :: [s_3]$$
$$X2[s_1] = s_1 :: (X1[s_1] \oplus R)$$
$$R[s_3] \quad = s_3 :: [s_4] \quad \text{(generated while } X2[s_1] \text{ is solved)}$$

Each $X_i[s_j] = E_{ij}$ is *a first-order equation* whose answer is a set of parse stacks.

The equations for $X1[s_1]$ and $X2[s_1]$ are mutually recursively defined, and their solutions are computed by least-fixed-point iteration. Here are the solutions:

$$X3(s_0) = X1[s_0] = [s_5]$$
$$X1[s_0] = X0[s_0] \cup X2[s_0] = [s_5] \cup [s_5] = [s_5]$$
$$X0(s_0) = [s_5]$$
$$X2[s_0] = s_0 :: (X1[s_1] \oplus R) \Rightarrow s_0 :: s_1 :: R[s_3] = s_0 :: s_1 :: s_3 :: [s_4] \Rightarrow [s_5]$$
$$R[s_3] \quad = s_3 :: [s_4]$$
$$X1[s_1] = X0[s_1] \cup X2[s_1] = (s_1 :: [s_3]) \cup (s_1 :: [s_3]) = s_1 :: [s_3]$$
$$X0[s_1] = s_1 :: [s_3]$$
$$X2[s_1] = s_1 :: (X1[s_1] \oplus R) = s_1 :: s_1 :: R[s_3] \Rightarrow s_1 :: [s_3]$$

$X3[s_0] = [s_5]$ validates that the strings printed at the hot spot must be S-phrases. (Note again: these answers are really sets, e.g., $X3[s_0] = \{[s_5]\}$.)

The algorithm that generates the equations and their solutions is a demand-driven data-flow analysis [1, 5, 6], similar to *minimal function-graph semantics* [7], computed by a worklist algorithm.
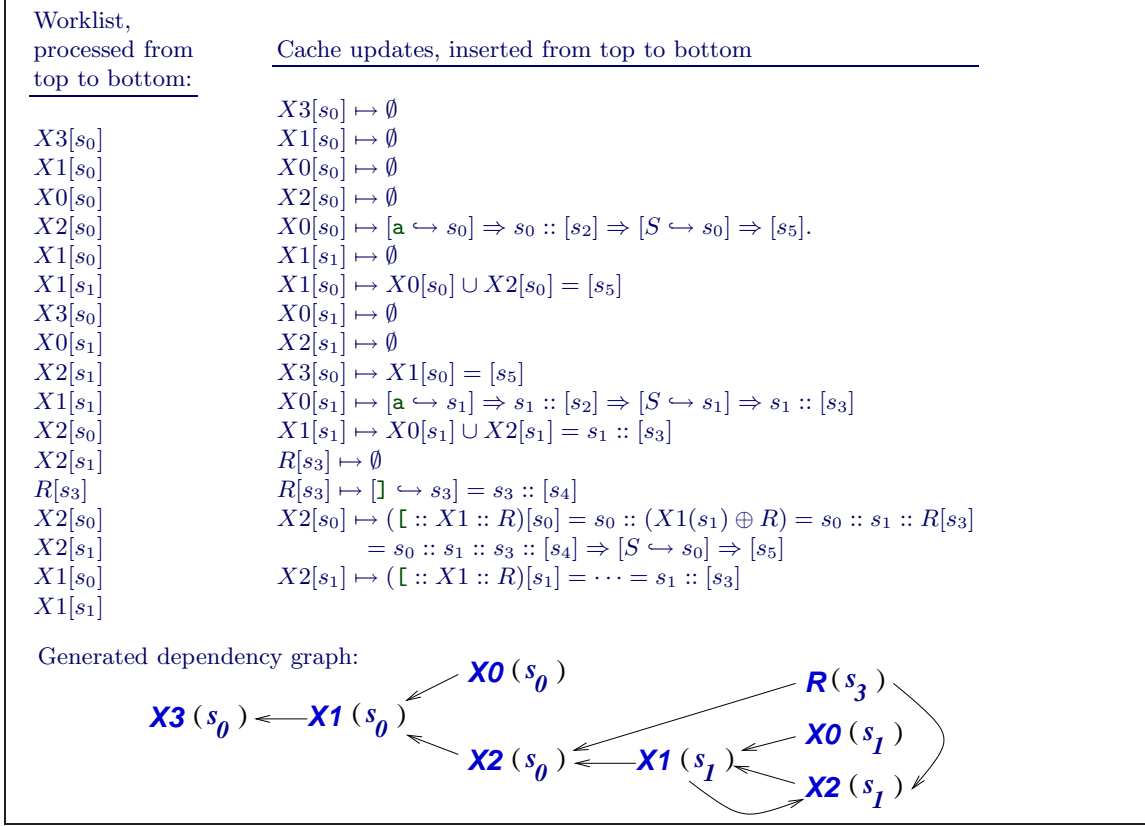
```
Worklist,
processed from        Cache updates, inserted from top to bottom
top to bottom:
                      X3[s_0] ↦ ∅
X3[s_0]               X1[s_0] ↦ ∅
X1[s_0]               X0[s_0] ↦ ∅
X0[s_0]               X2[s_0] ↦ ∅
X2[s_0]               X0[s_0] ↦ [a ↪ s_0] ⇒ s_0 :: [s_2] ⇒ [S ↪ s_0] ⇒ [s_5].
X1[s_0]               X1[s_1] ↦ ∅
X1[s_1]               X1[s_0] ↦ X0[s_0] ∪ X2[s_0] = [s_5]
X3[s_0]               X0[s_1] ↦ ∅
X0[s_1]               X2[s_1] ↦ ∅
X2[s_1]               X3[s_0] ↦ X1[s_0] = [s_5]
X1[s_1]               X0[s_1] ↦ [a ↪ s_1] ⇒ s_1 :: [s_2] ⇒ [S ↪ s_1] ⇒ s_1 :: [s_3]
X2[s_0]               X1[s_1] ↦ X0[s_1] ∪ X2[s_1] = s_1 :: [s_3]
X2[s_1]               R[s_3] ↦ ∅
R[s_3]                R[s_3] ↦ [] ↪ s_3] = s_3 :: [s_4]
X2[s_0]               X2[s_0] ↦ ([ :: X1 :: R)[s_0] = s_0 :: (X1(s_1) ⊕ R) = s_0 :: s_1 :: R[s_3]
X2[s_1]                        = s_0 :: s_1 :: s_3 :: [s_4] ⇒ [S ↪ s_0] ⇒ [s_5]
X1[s_0]               X2[s_1] ↦ ([ :: X1 :: R)[s_1] = ⋯ = s_1 :: [s_3]
X1[s_1]
```

Generated dependency graph:

Fig. 3. Worklist-algorithm calculation of call, $X3[s_0]$, in Figure 1

The initialization step places initial call, $X0[s_0]$, into the worklist and into the dependency graph and assigns to the cache the partial solution, $Cache[X0[s_0]] = \emptyset$. The iteration step repeats the following until the worklist is empty:

Extract a call, $X[s]$, from the worklist, and for the corresponding flow equation, $X = E$, compute $E[s]$:

1. While computing $E[s]$, if a call, $X'[s']$ is encountered, *(i)* add the dependency, $X'[s'] \to X[s]$, to the dependency graph if not already present; *(ii)* if there is no entry for $X'[s']$ in the cache, then assign $Cache[X'[s']] = \emptyset$ to the cache and place $X'[s']$ on the worklist. *(iii)* use $Cache[X'[s']]$ as the meaning of $X'[s']$ in the computation of $E[s]$.
2. When $E[s]$ computes to an answer set, P, and P contains an abstract parse stack not already listed in $Cache[X[s]]$, then set $Cache[X[s]] = Cache[X[s]] \cup P$ and add to the worklist all $X''[s'']$ such that $X[s] \to X''[s'']$ appears in the dependency graph.

3   Abstract parse stacks

In the previous example, the result for each $X_i[s_j]$ was a single stack. In general, a set of parse stacks can result, e.g., for

```
x = '['                     X0 = [
while ...                    X1 = X0 ⊔ X2
  x = x . '['                X2 = X1 · [
  x = x . 'a' . ']'          X3 = X1 · a · ]
```

at conclusion, x holds zero or more left brackets and an S-phrase; $X3[s_0]$ is the infinite set, $\{[s_5],\ s_1 :: [s_3],\ s_1 :: s_1 :: [s_3],\ s_1 :: s_1 :: s_1 :: [s_3],\ \cdots\}$.

To bound the set, we abstract it by "folding" its stacks so that no parse state repeats in a stack; *this generates a subgraph of the LR-parse controller*.

For example, $p = s_1 :: [s_1]$ is a linked list, a graph, $\leftarrow s_1 \leftarrow s_1 \leftarrow$, where the stack's top and bottom are marked by pointers; when we push a state, e.g., $p :: [s_2]$, we get $\leftarrow s_1 \leftarrow s_1 \leftarrow s_2 \leftarrow$. The folded stack is formed by merging same-state objects and retaining all links: $\leftarrow s_1 \leftarrow s_2 \leftarrow$. (This can be written as the regular expression, $s_1^+ :: [s_2]$.) Folding can apply to multiple states, e.g.,

$\leftarrow s_6 \leftarrow s_7 \leftarrow s_6 \leftarrow s_7 \leftarrow s_6 \leftarrow s_8 \leftarrow$ folds to $\leftarrow s_6 \leftarrow s_7 \leftarrow s_8 \leftarrow$.

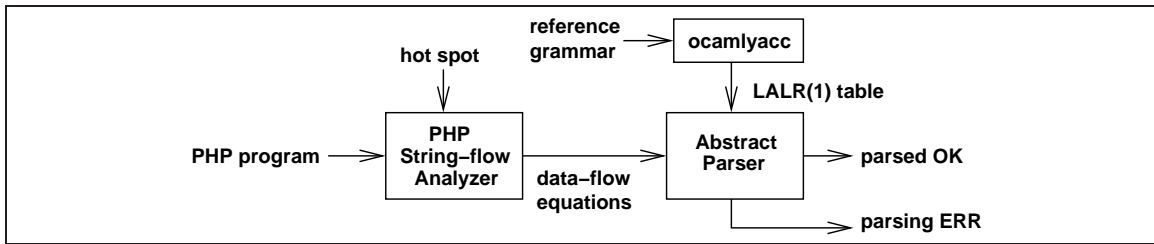For the above example, $X3[s_0] = \{[s_5],\ s_1^+ :: [s_3]\}$.

reference grammar → ocamlyacc

hot spot

LALR(1) table

PHP program → PHP String–flow Analyzer → data–flow equations → Abstract Parser → parsed OK

parsing ERR

Fig. 4. Implementation

4   Implementation and experiments

The implementation is in Objective Caml. The front end of Minamide's analyzer for PHP [8] was modified to accept a PHP program with a hot-spot location and to return data-flow equations with string operations for the hot spot. `ocamlyacc` produces an LALR(1) parsing table, and the abstract parser uses the data-flow equations and the parsing table to parse statically the strings generated by the PHP program. Abstract parsing works directly on characters (not tokens), so the reference grammar is written for scannerless parsing. (Performance was good enough for practical use.)

We applied our tool to a suite of PHP programs that dynamically generate HTML documents, the same one studied by Minamide [8], using a MacOSX with an Intel Core 2 Duo Processor (2.56GHz) and 4 GByte memory:

	webchess	faqforge	phpwims	timeclock	schoolmate
files	21	11	30	6	54
lines	2918	1115	6606	1006	6822
no. of hot spots	6	14	30	7	1
no. of parsings	6	16	36	7	19
parsed OK	5	1	19	0	1
parsed ERR	1	15	17	7	18
no. of alarms	1	31	16	14	20
true positives	1	31	13	14	17
false positives	0	0	3	0	3
time(sec)	0.224	0.155	1.979	0.228	2.077

We manually identified the hot spots and ran our abstract parser for each hot spot. Since we do not yet have parse-error recovery, each time a parse error was identified by our analyzer, we located the source of the error, fixed it, and tried again until no parse errors were detected.

All the false-positive alarms were caused by ignoring the tests within conditional commands. The parsing time shown in the table is the sum of all execution times needed to find all parsing errors for all hot spots. The reference grammar's parse table took 1.323 seconds to construct; this is not included in the analysis times.

	webchess	faqforge	phpwims	timeclock	schoolmate
files	21	11	30	6	54
lines	2918	1115	6606	1006	6822
no. of hot spots	6	14	30	7	1
no. of parsings	6	16	36	7	19
parsed OK	5	1	19	0	1
parsed ERR	1	15	17	7	18
no. of alarms	1	31	16	14	20
true positives	1	31	13	14	17
false positives	0	0	3	0	3
time(sec)	0.224	0.155	1.979	0.228	2.077

The alarms are classified below:

classification	occurrences
open/close tag syntax error	11
open/close tag missing	45
superfluous tag	5
improperly nested	14
misplaced tag	5
escaped character syntax error	2

All in all, our abstract parser works without limiting the nesting depth of tags, validates the syntax reasonably fast, and is guaranteed to find all parsing errors reducing inevitable false alarms to a minimum.

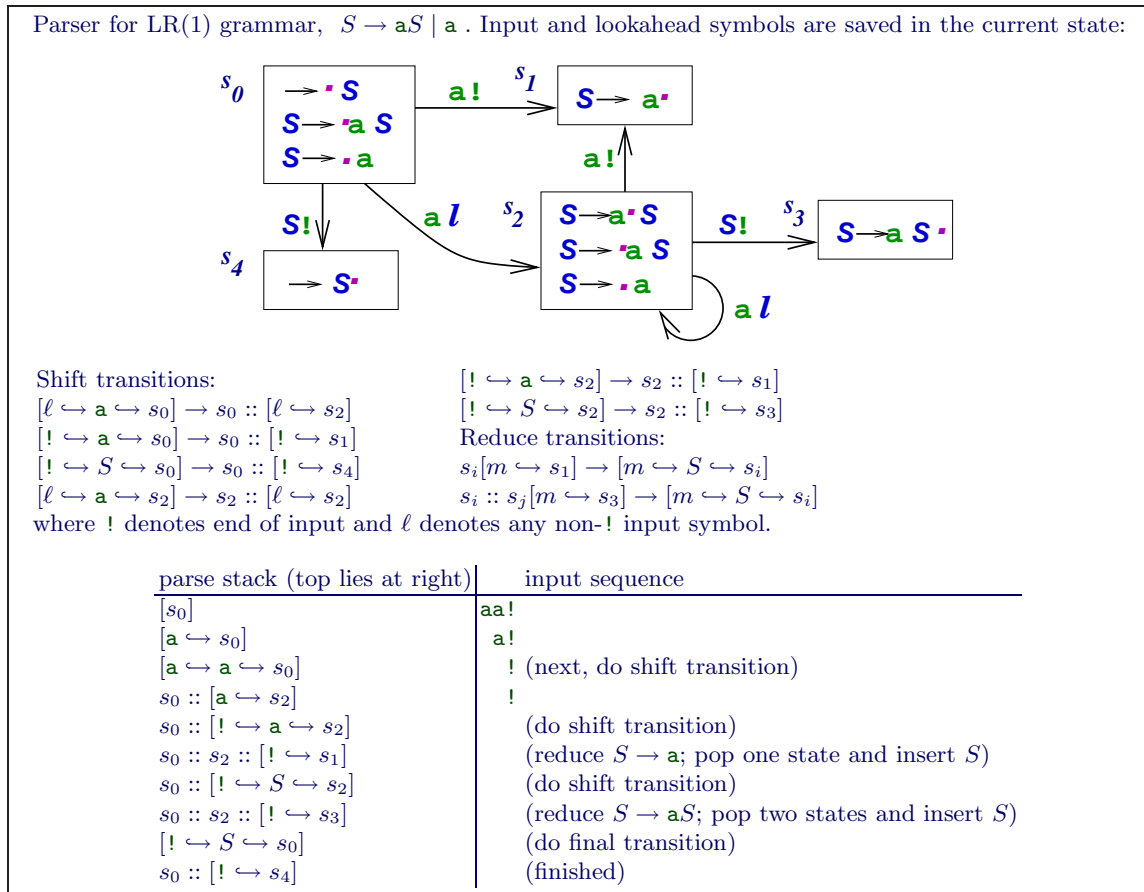## 5 LR(k) grammars are accommodated the same way

Parser for LR(1) grammar, $S \rightarrow \mathtt{a}S \mid \mathtt{a}$. Input and lookahead symbols are saved in the current state:

Shift transitions:

$[\ell \hookrightarrow \mathtt{a} \hookrightarrow s_0] \rightarrow s_0 :: [\ell \hookrightarrow s_2]$

$[! \hookrightarrow \mathtt{a} \hookrightarrow s_0] \rightarrow s_0 :: [! \hookrightarrow s_1]$

$[! \hookrightarrow S \hookrightarrow s_0] \rightarrow s_0 :: [! \hookrightarrow s_4]$

$[\ell \hookrightarrow \mathtt{a} \hookrightarrow s_2] \rightarrow s_2 :: [\ell \hookrightarrow s_2]$

$[! \hookrightarrow \mathtt{a} \hookrightarrow s_2] \rightarrow s_2 :: [! \hookrightarrow s_1]$

$[! \hookrightarrow S \hookrightarrow s_2] \rightarrow s_2 :: [! \hookrightarrow s_3]$

Reduce transitions:

$s_i[m \hookrightarrow s_1] \rightarrow [m \hookrightarrow S \hookrightarrow s_i]$

$s_i :: s_j[m \hookrightarrow s_3] \rightarrow [m \hookrightarrow S \hookrightarrow s_i]$

where ! denotes end of input and ℓ denotes any non-! input symbol.

parse stack (top lies at right)	input sequence
$[s_0]$	aa!
$[\mathtt{a} \hookrightarrow s_0]$	a!
$[\mathtt{a} \hookrightarrow \mathtt{a} \hookrightarrow s_0]$	! (next, do shift transition)
$s_0 :: [\mathtt{a} \hookrightarrow s_2]$	!
$s_0 :: [! \hookrightarrow \mathtt{a} \hookrightarrow s_2]$	(do shift transition)
$s_0 :: s_2 :: [! \hookrightarrow s_1]$	(reduce $S \rightarrow \mathtt{a}$; pop one state and insert S)
$s_0 :: [! \hookrightarrow S \hookrightarrow s_2]$	(do shift transition)
$s_0 :: s_2 :: [! \hookrightarrow s_3]$	(reduce $S \rightarrow \mathtt{a}S$; pop two states and insert S)
$[! \hookrightarrow S \hookrightarrow s_0]$	(do final transition)
$s_0 :: [! \hookrightarrow s_4]$	(finished)

Fig. 5. An LR(k) grammar uses a state of form, $[\ell_k \hookrightarrow \ell_{k-1} \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$.

When a program is statically parsed with an LR(k) grammar, $k > 0$, the generated equations have form,

$$Xi[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] = E$$

for $0 < j < k$. Alas, this means a residual-equation set of order $(k+1)!$.

6 Abstract parsing with string-replacement operations

Now, a parse state has form, $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, $0 < j < k$, where the ℓ_i are the inputs and s is the state of the parser automaton. A string update operation, e.g.,

```
y = replace 'aa' by 'b' in x
```

defines an automaton (more precisely, a *transducer*), too:
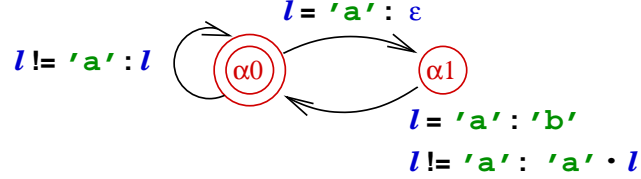
(We use $B : \ell$ to mean "take the transition if B holds true and emit letter ℓ as output".) When a `replace` operation appears in a program, *the transducer, α, defined by `replace` is composed with the parser automaton* — a state configuration now holds *two* states:

$$[\ell_{new} \hookrightarrow \alpha_m, \ell_k \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$$

Here, α_m is the current state of the transducer and s is the current state of the parser. A new input, ℓ_{new}, submits first to α_m, which transits and possibly emits input for s:

$$[\alpha_n, \ell_{k+1} \hookrightarrow \ell_k \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$$

In this way, strings are updated by `replace` before they are parsed. The assignment,

```
x = replace S1 by S2 in E
```

generates the flow equation

$$X = insert_\alpha \cdot E \cdot erase_\alpha$$

where α_0 names the start state of transducer α generated from patterns $S1$ and $S2$ and

$$insert_\alpha[\cdots s] = [\alpha_0, \cdots s]$$
$$erase_\alpha[\alpha_i, \cdots s] = [\cdots s]$$

So, the string generated from E goes into α_0, which processes it and emits output for s's input stream.

(Note: in the definition of $erase$, state α_i must be final. If not, the input symbols it holds are flushed and are added to s's input sequence.)

For example, the operator, `replace 'b' by 'a' in Y`, generates this automaton, β:

$$\ell = \mathsf{b}\colon \beta_0 \rightarrow \beta_0 : \mathsf{a}$$
$$\ell \neq \mathsf{b}\colon \beta_0 \rightarrow \beta_0 : \ell$$

For this script and its flow equations,

```
y = 'b'                                    Y = b
x = 'a'.(replace 'b' by 'a' in y)          X = a · (insertᵦ · Y · eraseᵦ)
```

$$Y = \mathsf{b}$$
$$X = \mathsf{a} \cdot (insert_\beta \cdot Y \cdot erase_\beta)$$

The abstract parse of $X \cdot \text{!}$ proceeds like this:

$$
\begin{aligned}
(X \cdot \text{!})[s_0] \quad &= X[s_0] \oplus \text{!} \\
X[s_0] \quad &= (\mathsf{a} \cdot insert_\beta \cdot Y \cdot erase_\beta)[s_0] \\
&= \mathsf{a}[s_0] \oplus (insert_\beta \cdot Y \cdot erase_\beta) \\
&= (insert_\beta \cdot Y \cdot erase_\beta)[\mathsf{a} \hookrightarrow s_0] \\
&= (Y \cdot erase_\beta)[\beta_0, \mathsf{a} \hookrightarrow s_0]) \\
Y[\beta_0, \mathsf{a} \hookrightarrow s_0] &= \mathsf{b}[\beta_0, \mathsf{a} \hookrightarrow s_0] \\
&= [\mathsf{b} \hookrightarrow \beta_0, \mathsf{a} \hookrightarrow s_0] \\
&= [\beta_0, \mathsf{a} \hookrightarrow \mathsf{a} \hookrightarrow s_0] \\
&= s_0 :: [\beta_0, \mathsf{a} \hookrightarrow s_2]
\end{aligned}
$$

Once all of Y's string is processed, automaton β is erased from the compound parse state:

$$
\begin{aligned}
X[s_0] &= (s_0 :: [\beta_0, \mathsf{a} \hookrightarrow s_2]) \oplus erase_\beta \\
&= s_0 :: erase_\beta[\beta_0, \mathsf{a} \hookrightarrow s_2] \\
&= s_0 :: [\mathsf{a} \hookrightarrow s_2]
\end{aligned}
$$

Our embedding of string-replacement operations into the parse state lets us retain the existing least-fixed point machinery for computing the solutions to the a script's flow equations. So, it is perfectly acceptable to allow string replacements within loop bodies — this surmounts existing techniques [2, 3, 8], because we are *not* generating a new grammar to approximate and check.
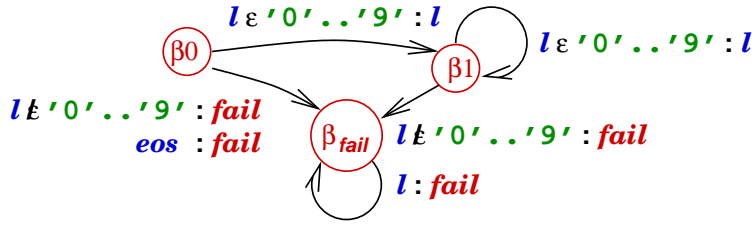
7 Using string-replacement automata to implement conditional tests

One fundamental technique needed for implementing taint analysis [11, 12, 10] is implementing filter functions for the tests of conditional commands. For example,

```
read x
if isAllDigits(x) :
then · · · the analysis assumes that x holds all digits · · ·
```

Think of the test expression, `isAllDigits(x)`, as an automaton (transducer) that reads the string contents of `x` and emits *failure* if a character of the input string is a nondigit. A failure causes the subsequent analysis to fail, too. In this fashion, the automaton acts as a "diode" or "filter function" that prevents non-digit string input from entering the conditional's body.

Here is the filter automaton for the test, `isAllDigits(x)`:

The filter automaton is a string-replacement automaton that emits *fail* when the input string does not satisfy the boolean test. The complement automaton, $\neg\beta$, merely swaps the outputs, ℓ and *fail*.

Our approach to analyzing conditional statements goes as follows:

For the conditional,
```
if B(x):
then · · · x · · ·
else · · · x · · ·
```

generate these flow equations:
$$X_B = insert_\beta \cdot X \cdot erase_\beta$$
$$\cdots X_B \cdots$$
$$X_{\neg B} = insert_{\neg\beta} \cdot X \cdot erase_{\neg\beta}$$
$$\cdots X_{\neg B} \cdots$$

where β is the automaton that implements test B and $\neg\beta$ implements $\neg B$.

The *fail* character is special — when it is processed as an input, it causes the parse itself to denote $\bot$ (empty set in the powerset lattice):

$$[\cdots, fail, \cdots] = \bot$$

For example,

```
x = 'a'                    X0 = a
if isAllDigits(x):         X1 = insert_β · X0 · erase_β
    print x !              X2 = X1 · !
```

and

$$X2[s_0] \quad\;\; = X1 \cdot ![s_0] = X1[s_0] \oplus !$$
$$X1[s_0] \quad\;\; = X0[\beta_0, s_0] \oplus erase_\beta$$
$$X0[\beta_0, s_0] = \mathsf{a}[\beta_0, s_0] = [\mathsf{a} \hookrightarrow \beta_0, s_0] = [\beta_0, fail \hookrightarrow s_0] = \bot$$

Hence,

$$X1[s_0] = erase_\beta \bot = \bot$$
$$X2[s_0] = \bot \oplus ! = \bot$$

The analysis correctly predicts that nothing prints within the body of the conditional.

8   Modelling user input with nonterminals and unfolding

Say that a module uses a string-valued global variable that is initialized outside of the module. If we assume the variable's value has the structure named by a nonterminal, then the global variable can be used in an abstract parse. For example, assume global variable **g** holds an S-structured string:

```
x = 'a'.g
print x !
```

$$G = S$$
$$X = \mathsf{a} \cdot G$$
$$X' = X \cdot !$$

We compute the abstract parse for $X'[s_0]$:

$$(\mathsf{a} \cdot G \cdot !)[s_0] = G[\mathsf{a} \hookrightarrow s_0] \oplus !$$
$$G[\mathsf{a} \hookrightarrow s_0] \quad = [S \hookrightarrow \mathsf{a} \hookrightarrow s_0] = s_0 :: [S \hookrightarrow s_2]$$

Hence,

$$G[\mathsf{a} \hookrightarrow s_0] \oplus ! = s_0 :: ![S \hookrightarrow s_2] = s_0 :: [! \hookrightarrow S \hookrightarrow s_2]$$
$$= s_0 :: s_2 :: [! \hookrightarrow s_3] = [! \hookrightarrow S \hookrightarrow s_0]$$
$$= s_0 :: [! \hookrightarrow s_4]$$

In a similar way, user input, supplied via **read** commands, can be assumed to have structure named by a nonterminal, and abstract parsing can be undertaken:

```
g = read_S()
x = 'a'.g
print x !
```

$$G = S$$
$$X = \mathsf{a} \cdot G$$
$$X' = X \cdot !$$

This proceeds just like the previous example. (Of course, we must supply a script that parses the input at runtime, to ensure that the input assumption is not violated.)

But there is a rub — say that the script includes string-replacement operations, which cannot process nonterminals. We solve this problem by unfolding the nonterminal, supplying the generated strings to the string-replacement automaton:

```
x = read_S()
y = replace 'aa' by 'a' in x
print y !
```

$$X = S$$
$$S = \mathsf{a} \cdot S \sqcup \mathsf{a}$$
$$Y = insert_\gamma \cdot X \cdot erase_\gamma$$
$$Y' = Y \cdot !$$

where automaton γ is defined,

$$\ell = \mathsf{a}\colon \gamma_0 \to \gamma_1 : \epsilon \qquad\qquad \ell \neq \mathsf{a}\colon \gamma_1 \to \gamma_0 : \mathsf{a} \cdot \ell$$
$$\ell \neq \mathsf{a}\colon \gamma_0 \to \gamma_0 : \ell \qquad\qquad \ell = eos\colon \gamma_0 \to \gamma_0 : \epsilon$$
$$\ell = a\colon \gamma_1 \to \gamma_0 : \mathsf{a} \qquad\qquad \ell = eos\colon \gamma_1 \to \gamma_0 : \mathsf{a}$$

where γ_0 is the final state.

The analysis of the **print** command generates these first-order equations to solve:

$$Y'[s_0] \quad = Y[s_0] \oplus !$$
$$Y[s_0] \quad = (X[\gamma_0, s_0]) \oplus erase_\gamma$$
$$X[\gamma_0, s_0] = S[\gamma_0, s_0]$$

The call to S generates these equations, which explain how to replace and parse all strings generated from nonterminal, S:

$$
\begin{aligned}
S[\gamma_0, s_0] &= (\mathsf{a} \cdot S)[\gamma_0, s_0] \cup \mathsf{a}[\gamma_0, s_0] = [\mathsf{a} \hookrightarrow \gamma_0, s_0] \oplus S \cup [\mathsf{a} \hookrightarrow \gamma_0, s_0] \\
&= [\gamma_1, s_0] \oplus S \cup [\gamma_1, s_0] \\
&= S[\gamma_1, s_0] \cup [\gamma_1, s_0] \\
S[\gamma_1, s_0] &= (\mathsf{a} \cdot S)[\gamma_1, s_0] \cup \mathsf{a}[\gamma_1, s_0] = S[\gamma_0, \mathsf{a} \hookrightarrow s_0] \cup [\gamma_0, \mathsf{a} \hookrightarrow s_0] \\
S[\gamma_0, \mathsf{a} \hookrightarrow s_0] &= S[\gamma_1, \mathsf{a} \hookrightarrow s_0] \cup [\gamma_1, \mathsf{a} \hookrightarrow s_0] \\
S[\gamma_1, \mathsf{a} \hookrightarrow s_0] &= [\gamma_0, \mathsf{a} \hookrightarrow \mathsf{a} \hookrightarrow s_0] \oplus S \cup [\gamma_0, \mathsf{a} \hookrightarrow \mathsf{a} \hookrightarrow s_0] \\
&= s_0 :: S[\gamma_0, \mathsf{a} \hookrightarrow s_2] \cup s_0 :: [\gamma_0, \mathsf{a} \hookrightarrow s_2] \\
S[\gamma_0, \mathsf{a} \hookrightarrow s_2] &= S[\gamma_1, \mathsf{a} \hookrightarrow s_2] \cup [\gamma_1, \mathsf{a} \hookrightarrow s_2] \\
S[\gamma_1, \mathsf{a} \hookrightarrow s_2] &= s_2 :: S[\gamma_0, \mathsf{a} \hookrightarrow s_2] \cup s_2 :: [\gamma_0, \mathsf{a} \hookrightarrow s_2]
\end{aligned}
$$

All reachable combinations of the string-replacement automaton and parse controller are generated. This completes the equation set, which is solved in the usual way.

The state explosion that is typical in such examples can be controlled by using SLR(k) or LALR(k) grammars to define string structure.

With the technique just illustrated, we can show the correctness of input-validation codings. For example, a script that goes

```
x = read_S()
if isAllDigits(x):
then···
```

can be analyzed with respect to the automaton defined by `isAllDigits` and this reference grammar:

$$
\begin{aligned}
S &::= C \mid CS \\
C &::= D \mid N \\
D &::= 0 \cdots 9 \\
N &::= \text{ all characters not in } D
\end{aligned}
$$

From here, it is only a small step to analyzing string-replacement and conditional-test automata to check for language inclusion, that is, all strings generated by a grammar nonterminal are accepted by the automaton.

9 Abstract semantic processing

Since we can predict the syntax of dynamically generated strings, we can predict their semantics as well by adapting attribute-grammar techniques. For example, binary numerals are generated by this LR(1) grammar,

$$B \rightarrow D \, B \mid D$$
$$D \rightarrow \mathtt{0} \mid \mathtt{1}$$

The semantics of binary numerals can be specified with *attributes* associated with the grammar symbols and *semantics rules* associated with the productions. The semantic rules specify how to calculate the parity of a numeral:

production	semantic rule
$\rightarrow B\,!$	answer $= B.even$
$B \rightarrow D\,B_1$	$B.even = B_1.even$
$B \rightarrow D$	$B.even = D.even$
$D \rightarrow \mathtt{0}$	$D.even = true$
$D \rightarrow \mathtt{1}$	$D.even = false$

The semantic rules can be computed during LR-parsing, as seen in Figure 6. When a reduce transition occurs, its corresponding semantic rule is computed. The computed result is annotated to its corresponding state, shown as a superscript in our notation. In the example in Figure 6, the computed attribute values are annotated only to the states, s_0 and s_3. For the example binary numeral, $\mathtt{101}$, the computed result is *false*, as expected.

Shift transitions:

$[\ell \hookrightarrow 0 \hookrightarrow s_0] \Rightarrow s_0 :: [\ell \hookrightarrow s_1]$

$[\ell \hookrightarrow 1 \hookrightarrow s_0] \Rightarrow s_0 :: [\ell \hookrightarrow s_2]$

$[\ell \hookrightarrow D \hookrightarrow s_0] \Rightarrow s_0 :: [\ell \hookrightarrow s_3]$

$[! \hookrightarrow D \hookrightarrow s_0] \Rightarrow s_0 :: [! \hookrightarrow s_5]$

$[! \hookrightarrow B \hookrightarrow s_0^b] \Rightarrow [s_6^b]$

$[\ell \hookrightarrow 0 \hookrightarrow s_3] \Rightarrow s_3 :: [\ell \hookrightarrow s_1]$

$[\ell \hookrightarrow 1 \hookrightarrow s_3] \Rightarrow s_3 :: [\ell \hookrightarrow s_2]$

$[\ell \hookrightarrow D \hookrightarrow s_3] \Rightarrow s_3 :: [\ell \hookrightarrow s_3]$

$[! \hookrightarrow D \hookrightarrow s_3] \Rightarrow s_3 :: [! \hookrightarrow s_5]$

$[! \hookrightarrow B \hookrightarrow s_3] \Rightarrow s_3 :: [! \hookrightarrow s_4]$

Reduce transitions:

$s :: [\ell \hookrightarrow s_1] \Rightarrow [\ell \hookrightarrow D \hookrightarrow s^{true}]$

$s :: [\ell \hookrightarrow s_2] \Rightarrow [\ell \hookrightarrow D \hookrightarrow s^{false}]$

$s_i :: s_j^b :: [\ell \hookrightarrow s_4] \Rightarrow [\ell \hookrightarrow B \hookrightarrow s_i^b]$

$s_i^b :: [\ell \hookrightarrow s_5] \Rightarrow [\ell \hookrightarrow B \hookrightarrow s_i^b]$

where ! denotes end of input and ℓ denotes any non-! input symbol.

parse stack (top lies at right)	input sequence
$[s_0]$	101!
$[1 \hookrightarrow s_0]$	01!
$[0 \hookrightarrow 1 \hookrightarrow s_0]$	1! (shift)
$s_0 :: [0 \hookrightarrow s_2]$	1! (reduce $D \to 1$)
$s_0 :: [D \hookrightarrow s_0^{false}]$	1! (shift)
$s_0^{false} :: [0 \hookrightarrow s_3]$	1!
$s_0^{false} :: [1 \hookrightarrow 0 \hookrightarrow s_3]$	! (shift)
$s_0^{false} :: s_3 :: [1 \hookrightarrow s_1]$	! (reduce $D \to 0$)
$s_0^{false} :: [1 \hookrightarrow D \hookrightarrow s_3^{true}]$	! (shift)
$s_0^{false} :: s_3^{true} :: [1 \hookrightarrow s_3]$	!
$s_0^{false} :: s_3^{true} :: [! \hookrightarrow 1 \hookrightarrow s_3]$	(shift)
$s_0^{false} :: s_3^{true} :: s_3 :: [! \hookrightarrow s_2]$	(reduce $D \to 1$)
$s_0^{false} :: s_3^{true} :: [! \hookrightarrow D \hookrightarrow s_3^{false}]$	(shift)
$s_0^{false} :: s_3^{true} :: s_3^{false} :: [! \hookrightarrow s_5]$	(reduce $B \to D$)
$s_0^{false} :: s_3^{true} :: [! \hookrightarrow B \hookrightarrow s_3^{false}]$	(shift)
$s_0^{false} :: s_3^{true} :: s_3^{false} :: [! \hookrightarrow s_4]$	(reduce $B \to D\,B$)
$s_0^{false} :: [! \hookrightarrow B \hookrightarrow s_3^{false}]$	(shift)
$s_0^{false} :: s_3^{false} :: [! \hookrightarrow s_4]$	(reduce $B \to D\,B$)
$[! \hookrightarrow B \hookrightarrow s_0^{false}]$	(shift)
$[s_6^{false}]$	(finished)

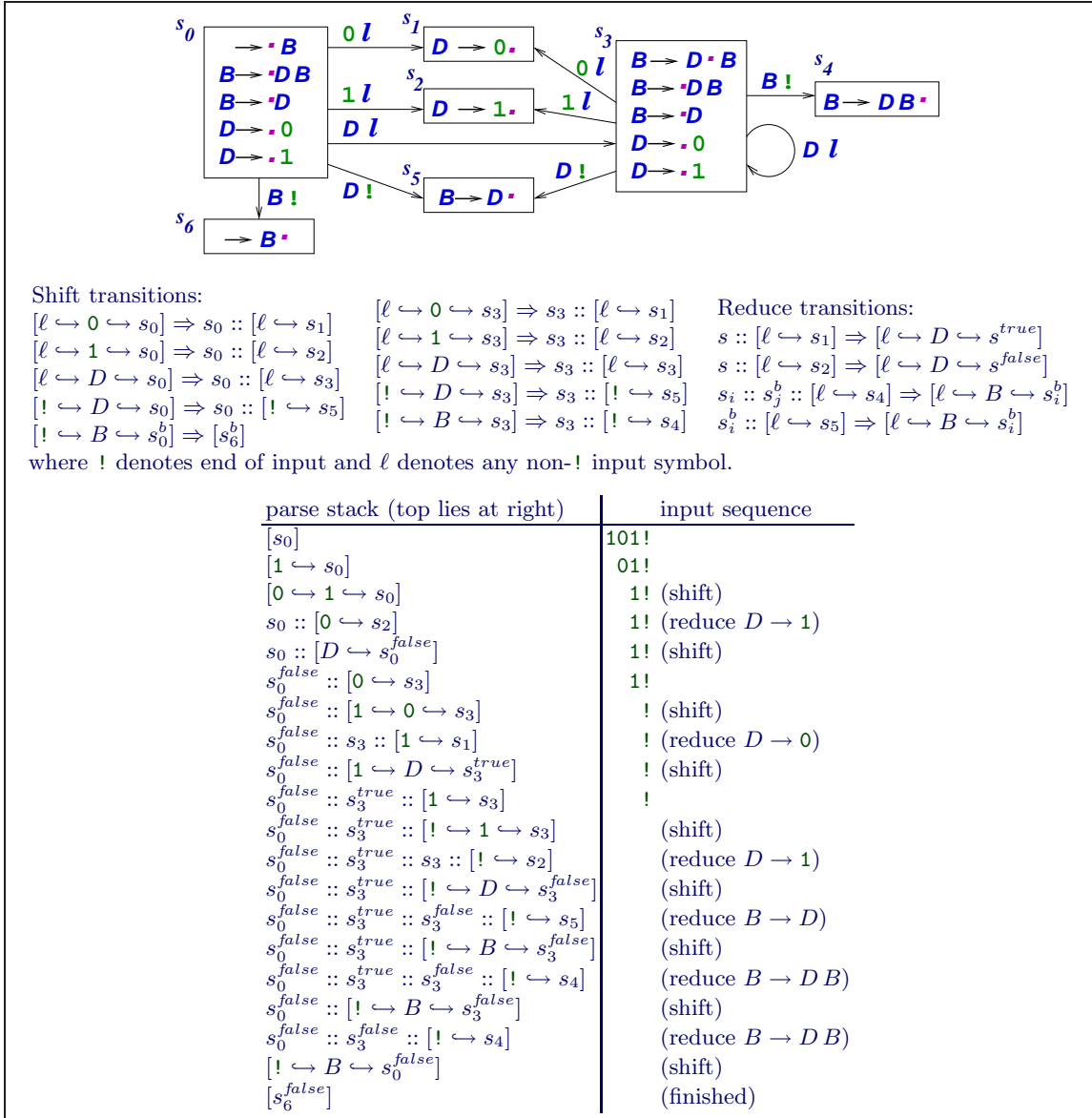**Fig. 6.** Syntax-directed semantic processing for LR(1) grammar, $B \to DB \mid D$, $D \to 0 \mid 1$

For this program,

```
x = '0'                         X0 = 0
while ...                       X1 = X0 ⊔ X2
    x = read_D() · x            X2 = D · X1
    print x · !                 X3 = X1 · !
```

$$X0 = 0$$
$$X1 = X0 \sqcup X2$$
$$X2 = D \cdot X1$$
$$X3 = X1 \cdot {!}$$

its abstract parsing with synthesized-attribute computing proceeds as follows:

$$
\begin{aligned}
X3[s_0] \quad &= (X1 \cdot {!})[s_0] = X1[s_0] \oplus {!} \\
X1[s_0] \quad &= X0[s_0] \cup X2[s_0] \\
X0[s_0] \quad &= 0[s_0] = \{[0 \hookrightarrow s_0]\} \\
X2[s_0] \quad &= (D \cdot X1)[s_0] = [D \hookrightarrow s_0] \oplus X1 = X1[D \hookrightarrow s_0] \\
&= X0[D \hookrightarrow s_0] \cup X2[D \hookrightarrow s_0] \\
X0[D \hookrightarrow s_0] &= \{[0 \hookrightarrow D \hookrightarrow s_0]\} \Rightarrow \{s_0 :: [0 \hookrightarrow s_3]\} \\
X2[D \hookrightarrow s_0] &= (D \cdot X1)[D \hookrightarrow s_0] = [D \hookrightarrow D \hookrightarrow s_0] \oplus X1 \\
&\Rightarrow \{s_0 :: [D \hookrightarrow s_3]\} \oplus X1 = \{s_0 :: X1[D \hookrightarrow s_3]\} \\
X1[D \hookrightarrow s_3] &= X0[D \hookrightarrow s_3] \cup X2[D \hookrightarrow s_3] \\
X0[D \hookrightarrow s_3] &= \{[0 \hookrightarrow D \hookrightarrow s_3]\} \Rightarrow \{s_3 :: [0 \hookrightarrow s_3]\} \\
X2[D \hookrightarrow s_3] &= (D \cdot X1)[D \hookrightarrow s_3] = [D \hookrightarrow D \hookrightarrow s_3] \oplus X1 \\
&\Rightarrow \{s_3 :: [D \hookrightarrow s_3]\} \oplus X1 = \{s_3 :: X1[D \hookrightarrow s_3]\}
\end{aligned}
$$

Now we have a recursive equation to solve:

$$
\begin{aligned}
X1[D \hookrightarrow s_3] &= \{s_3 :: [0 \hookrightarrow s_3]\} \cup \{s_3 :: X1[D \hookrightarrow s_3]\} \\
&= \{s_3^+ :: [0 \hookrightarrow s_3]\}
\end{aligned}
$$

Using this result, we obtain:

$$
\begin{aligned}
X2[D \hookrightarrow s_0] &= \{s_0 :: s_3^+ :: [0 \hookrightarrow s_3]\} \\
X2[s_0] &= \{s_0 :: [0 \hookrightarrow s_3], s_0 :: s_3^+ :: [0 \hookrightarrow s_3]\} \\
X1[s_0] &= \{[0 \hookrightarrow s_0], s_0 :: [0 \hookrightarrow s_3], s_0 :: s_3^+ :: [0 \hookrightarrow s_3]\} \\
X3[s_0] &= \{[{!} \hookrightarrow 0 \hookrightarrow s_0], s_0 :: [{!} \hookrightarrow 0 \hookrightarrow s_3], s_0 :: s_3^+ :: [{!} \hookrightarrow 0 \hookrightarrow s_3]\} \\
&= \{[s_6^{true}]\}
\end{aligned}
$$

since $\{[{!} \hookrightarrow 0 \hookrightarrow s_0]\} \Rightarrow \{s_0 :: [{!} \hookrightarrow s_1]\} \Rightarrow \{[{!} \hookrightarrow D \hookrightarrow s_0^{true}]\}$
$\qquad \Rightarrow \{s_0^{true} :: [{!} \hookrightarrow s_5]\} \Rightarrow \{[{!} \hookrightarrow B \hookrightarrow s_0^{true}]\} \Rightarrow \{[s_6^{true}]\}$
and $\{s_0 :: [{!} \hookrightarrow 0 \hookrightarrow s_3]\} \Rightarrow \{s_0 :: s_3 :: [{!} \hookrightarrow s_1]\} \Rightarrow \{s_0 :: [{!} \hookrightarrow D \hookrightarrow s_3^{true}]\}$
$\qquad \Rightarrow \{s_0 :: s_3^{true} :: [{!} \hookrightarrow s_5]\} \Rightarrow \{s_0 :: [{!} \hookrightarrow B \hookrightarrow s_3^{true}]\}$
$\qquad \Rightarrow \{s_0 :: s_3^{true} :: [{!} \hookrightarrow s_4]\} \Rightarrow \{[{!} \hookrightarrow B \hookrightarrow s_0^{true}]\} \Rightarrow \{[s_6^{true}]\}$
and $\{s_0 :: s_3^+ :: [{!} \hookrightarrow 0 \hookrightarrow s_3]\} \Rightarrow \{s_0 :: s_3^+ :: s_3 :: [{!} \hookrightarrow s_1]\}$
$\qquad \Rightarrow \{s_0 :: s_3^+ :: [{!} \hookrightarrow D \hookrightarrow s_3^{true}]\} \Rightarrow \{s_0 :: s_3^+ :: s_3^{true} :: [{!} \hookrightarrow s_5]\}$
$\qquad \Rightarrow \{s_0 :: s_3^+ :: [{!} \hookrightarrow B \hookrightarrow s_3^{true}]\} \Rightarrow \{s_0 :: s_3^+ :: s_3^{true} :: [{!} \hookrightarrow s_4]\}$
$\qquad \Rightarrow \{s_0 :: [{!} \hookrightarrow B \hookrightarrow s_3^{true}], s_0 :: s_3^+ :: [{!} \hookrightarrow B \hookrightarrow s_3^{true}]\}$
$\qquad$ (second set element adds nothing to fixed point)
$\qquad \Rightarrow \{s_0 :: s_3^{true} :: [{!} \hookrightarrow s_4]\} \Rightarrow \{[{!} \hookrightarrow B \hookrightarrow s_0^{true}]\} \Rightarrow \{[s_6^{true}]\}$

This proves that all possible string values of x at the end are well-structured B-phrases and even-valued. The approach is well suited to "type checking" XML-like documents; this application is currently under investigation.

10 Definitions of parsing and collecting semantics

An LR(k) *parse-stack configuration* is a sequence, $s_0 :: s_1 :: \cdots s_i :: [\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, $0 < j < k$, where $s_0 \cdots s_i, s$ are states from the parser controller; ℓ_0 is the input symbol; and $\ell_1 \cdots \ell_j$ are the lookahead symbols. $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$ is the *parse state* and will always be presented as the "top" of the parse-stack configuration.

A parse of input symbols $a_1 \cdots a_n!$ is defined as $[\![a_1 \cdots a_n!]\!][s_0]$, where s_0 is the parse controller's start state. Let c stand for a parse state. The transition rules in Figure 5 can be formalized as

$$[\![a]\!][\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] = move([a \hookrightarrow \ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s])$$

$$[\![E1 \cdot E2]\!]c = move([\![E1]\!]c \oplus [\![E2]\!])$$
$$\text{where } (s_0 :: s_1 :: \cdots s_i :: c') \oplus F = s_0 :: s_1 :: \cdots s_i :: F(c')$$

$move(s_0 :: \cdots :: s_i :: [\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]) =$
 if s is a final (reduce) state for grammar rule, $N \to U_1 U_2 \cdots U_m$, and $m \le n$,
 then return $move(s_0 :: \cdots :: s_{n-m} :: [\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow N \hookrightarrow s_{n-m+1}])$
  (pop top m states, and insert N at front of input stream)
 else if there is a match of $[\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$ to the left-hand-side of a transition rule,
 $[\ell_k \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] \to [\ell_k \hookrightarrow \cdots \hookrightarrow \ell_1 \hookrightarrow s']$,
 then return $move(s_0 :: s_1 :: \cdots s_i :: s :: [\ell_k \hookrightarrow \cdots \hookrightarrow \ell_1 \hookrightarrow s'])$
  (shift)
 else return $s_0 :: s_1 :: \cdots s_i :: [\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s]$, as is.

The next definition of interest is the semantics of the flow equations extracted from a script. A flow equation takes the form, $X = E$, where

$$E ::= a \mid E1 \cdot E2 \mid E1 \sqcup E2 \mid X_j$$

The semantics is called the *collecting semantics* and is defined like this:

$$[\![E]\!] : ParseState \to \mathcal{P}(ParseConfiguration)$$

$$[\![a]\!][\ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s] = \{move([a \hookrightarrow \ell_j \hookrightarrow \cdots \hookrightarrow \ell_0 \hookrightarrow s])\}$$

$$[\![E1 \cdot E2]\!]c = \{move(c') \mid c' \in [\![E1]\!]c \oplus [\![E2]\!]\}$$
$$\text{where } S \oplus F = \{tail(c) :: F(head(c)) \mid c \in S\}$$

$$[\![E1 \sqcup E2]\!]c = [\![E1]\!]c \cup [\![E2]\!]c$$

$$[\![X_j]\!]c = [\![E_j]\!]c, \text{ where } X_j = E_j \text{ is the corresponding flow equation}$$

The definition shows that sets can result from the calculation of the collecting semantics. See [4] for examples.

From the collecting semantics domain of sets of parse configurations, one defines an abstract interpretation by approximating a set of configurations by a finite set of finite configurations or by just a single configuration, say, written in regular-expression notation. This is developed in [4].

The resulting interpretation can be applied to a set of flow equations and solved with the usual least-fixed-point techniques. This yields *abstract parsing* of the strings generated by a script.

11　Conclusion

Injection and cross-site-scripting attacks can be reduced by analyzing the programs that dynamically generate documents [12]. We have improved the precision of such analyses by employing LR-parsing technology to validate the context-free grammatical structure of generated documents.

THANK YOU to Carolyn Talcott for your leadership, inspiration, and support!

This talk is saved at `www.cis.ksu.edu/∼schmidt/papers/hometalks.html`

References

1. G. Agrawal. Simultaneous demand-driven data-flow and call graph analysis. In *Proc. Int'l. Conf. Software Maintenance, Oxford*, 1999.
2. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Static analysis for dynamic XML. In *Proc. PLAN-X-02*, 2002.
3. A.S. Christensen, A. Møller, and M.I. Schwartzbach. Extending Java for high-level web service construction. *ACM TOPLAS*, 25, 2003.
4. K.-G. Doh, H. Kim, and D.A. Schmidt. Abstract parsing: static analysis of dynamically generated string output using lr-parsing technology. In *Proc. Static Analysis Symposium*. Springer LNCS 5673, 2009.
5. E. Duesterwald, R. Gupta, and M.L. Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM TOPLAS*, 19:992–1030, 1997.
6. S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symp. Foundations of Software Engg.*, 1995.
7. N.D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proc. 13th Symp. POPL*, pages 296–306. ACM Press, 1986.
8. Y. Minamide. Static approximation of dynamically generated web pages. In *Proc. 14th ACM Int'l Conf. on the World Wide Web*, pages 432–441, 2005.
9. P. Thiemann. Grammar-based analysis of string expressions. In *Proc. ACM workshop Types in languages design and implementation*, pages 59–70, 2005.
10. G. Wassermann, C. Gould, Z. Su, and P. Devanbu. Static checking of dymanically generated queries in database applications. *ACM Trans. Software Engineering and Methodology*, 16(4):14:1–27, 2007.
11. G. Wassermann and Z. Su. The essence of command injection attacks in web applications. In *Proc. 33d ACM Symp. POPL*, pages 372–382, 2006.
12. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. ACM PLDI*, pages 32–41, 2007.