BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

PARSING OF CONTEXT SENSITIVE LANGUAGES

TJD

Brno, January 2007

Lukáš Rychnovský

Abstract

This work presents some ideas from parsing Context-Sensitive languages. Introduces Scattered-Context grammars and languages and describes usage of such grammars to parse CS languages. Also there are presented additional results from type checking and formal program verification using CS parsing.

Keywords

Turing Machines, Parsing of Context-Sensitive Languages, Formal Program Verification, Scattered-Context Grammars.

Contents

1 Preface

This work results from [Kol–04] where relationship between regulated pushdown automata (RPDA) and Turing machines is discussed. We are particulary interested in algorithms which may be used to obtain RPDAs from equivalent Turing machines. Such interest arises purely from practical reasons because RPDAs provide easier implementation techniques for problems where Turing machines are naturally used. As a representant of such problems we study context-sensitive extensions of programming language grammars which are usually context-free. By introducing context-sensitive syntax analysis into the source code parsing process a whole class of new problems may be solved at this stage of a compiler. Namely issues with correct variable definitions.

In the second chapter we provide necessary definitions, such as Turing machine or the Chomsky hierarchy. The third chapter provides some possibilities for further development.

2 Preliminaries

This chapter presents basic definitions and theorems which are subsequently required. Basic insight into language theory is assumed.

Definition 2.1. A deterministic poshdown automaton (PA) is a 7-tuple $T = (Q, \Sigma, \Omega, \delta, s, \nabla, F)$, where

1. Q is a finite set of states.
2. Σ is a finite set of the input alphabet.
3. Ω is a finite set of the stack alphabet.
4. δ is a finite transition relation $(Q \times (\Sigma \cup \{\epsilon\}) \times \Omega)$ to $Q \times \Omega^*$.
5. s $\in$ Q; the start state.
6. $\nabla \in \Omega$ is the initial stack symbol
7. F $\subseteq$ Q, consisting of the final states.

Definition 2.2. A configuration of the PA is a triple (q, w, γ), where $q \in Q$ is current state, $w \in \Sigma^*$ are non read characters and $\gamma \in \Omega^*$ are symbols on stack.

Definition 2.3. A deterministic Turing machine (DTM) is a 5-tuple $T = (Q, \Sigma, \Gamma, q_0, \delta)$, where

1. Q is a finite set of states, assumed not to contain the halt state (q_F).
2. Σ, the input alphabet, is a set of symbols, Σ is assumed not to contain Δ, the blank symbol
3. Γ, the tape alphabet, is a finite set with $\Sigma \subseteq \Gamma$.
4. $q_0 \in Q$ (the initial state).
5. δ is a partial function from $Q \times \Gamma$ to $(Q \cup \{q_F\}) \times \Gamma \times \{R, L, S\}$.

Definition 2.4. A configuration of the DTM is a pair $(Q, x\underline{a}y))$ where q is a state, $x, y \in \Gamma^*, a \in \Gamma$, and the underlined symbol represents current position of the head, which allows to read from and write to a tape one symbol to the square of current position and which can possibly stay (S) in the same position, move right (R), or move left (L). We say

$$(q, x\underline{a}y) \vdash_T (r, z\underline{b}w)$$

if T passes from the configuration on the left to that on the right in one move and

$$(q, x\underline{a}y) \vdash_T^* (r, z\underline{b}w)$$

if T passes from the first configuration in zero or more moves.

Definition 2.5. An input string $x \in \Sigma^*$ is accepted by Turing machine T if starting T with an input x leads eventually to halting a configuration. In other words, x is accepted if for some strings $y, z \in \Gamma^*$ and some $a \in \Gamma$

$$(q_0, \underline{\Delta} x) \vdash_T^* (q_F, y\underline{a}z)$$

In this situation we say T halts on the input x. The language, accepted by T, is the set of input strings that are accepted by T.

Definition 2.6. A linear bounded Turing machine (LBTM) can move in both directions when scanning the input word. Furthermore, it is able to replace the scanned symbol with another symbol. While doing so it may use as much space only as is occupied by the input word.

Remark 1. For further details on linear bounded Turing machines see [Sal–73].

Definition 2.7. The Chomsky hierarchy defines a hierarchy of languages characterized by their defining grammars. It consists of the following levels:

1. Type-0 languages correspond to unrestricted grammars (this level includes all formal grammars). They generate exactly all languages that are recognized by a Turing machine. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which are recognized by an always halting Turing machine.

2. Type-1 languages correspond to context-sensitive grammars. These grammars have rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \rightarrow \epsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a non-deterministic Turing machine whose tape is bounded by a constant times the length of the input.

3. Type-2 languages correspond to context-free grammars. These are defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and non-terminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context free languages are the theoretical basis for the syntax of most programming languages.

4. Type-3 languages correspond to regular grammars. Such grammars restrict their rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal. The rule $S \rightarrow \epsilon$ is also here allowed if S does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages is obtained as the family of languages accepted by regular expressions. Consequently these languages are commonly used to define search patterns and the lexical structure of programming languages.

Definition 2.8. A grammar G is called linear if and only if its productions are of the two forms $X \to \alpha_1 Y \alpha_2$ and $X \to \alpha$ where X and Y are nonterminals and the α's are words over the terminal alphabet. The grammar G is called left-linear (resp. right-linear) if each of the words α_1 (resp. α_2) is empty.

Languages defined by linear grammars are called linear languages (denoted as LIN).

2.1 Scattered context grammars

Definition 2.9. A scattered context grammar (SCG) G is a quadruple (V, T, P, S), where V is a finite set of symbols, $T \subset V$, $S \in V \backslash T$, and P is a set of production rules of the form

$$(A_1, \ldots, A_n) \to (w_1, \ldots, w_n), n \geq 1, \forall A_i : A_i \in V \backslash T, \forall w_i : w_i \in V^*$$

Definition 2.10. Let $G = (V, T, P, S)$ be a SCG. Let G be a left derivating SCG then for

$$(A_1, \ldots, A_n) \to (w_1, \ldots, w_n) \in P$$

we write $x_1 A_1 x_2 A_2 \ldots x_n A_n x_{n+1} \Rightarrow x_1 w_1 x_2 w_2 \ldots x_n w_n x_{n+1}, x_i \in V^*$,if $x_1 \neq y_1 A_1 z_1$, $x_2 \neq y_2 A_2 z_2, \ldots x_n \neq y_n A_n z_n, y_i, z_i \in V^*$.

Definition 2.11. Let $G = (V, T, P, S)$ be a SCG. Let $(A_1, ..., A_n) \to (w_1, ..., w_n) \in P$. Then we define a derivation relation $\Rightarrow$ as follows: for $1 \leq i \leq n+1$ let $x_i \in V^*$. Then

$$x_1 A_1 x_2 A_2 ... x_n A_n x_{n+1} \Rightarrow x_1 w_1 x_2 w_2 ... x_n w_n x_{n+1}$$

$\Rightarrow^*$ is reflexive, transitive closure of $\Rightarrow$.
Language generated by the grammar G is defined as $L(G) = \{w \in T^* | S \Rightarrow^* w\}$.

Theorem 2.1. *Let $L_n(SC)$ be the family of languages generated by SC grammars whose number of productions that contains two or more context-free productions (degree of context sensitivity) is n or less. $L(RE)$ denotes the family of recursively enumerable languages.*

$$L_2(SC) = L_\infty(SC) = L(RE)$$

Proof. See [Med–03] Lemma 1 and Theorem 3. □

Remark 2. For further details on Turing machines see [Sal–73].

2.2 ZAP03

As an example of a context-free language we use a language called ZAP03 [ZAP–03] which has very similar syntax to Pascal. The ZAP03 grammar is

$s \rightarrow program$

$program \rightarrow proc$

$program \rightarrow fun$

$program \rightarrow gldec$

$gldec \rightarrow [global].dcl.[;].next$

$dcl \rightarrow typ.[:].[id].id\_list$

$id\_list \rightarrow [,].[id].id\_list$

$id\_list \rightarrow \epsilon$

$next \rightarrow gldec$

$next \rightarrow proc$

$next \rightarrow typ.next2$

$next2 \rightarrow [:].[id].id\_list.[;].next$

$next2 \rightarrow [id].parametr.[=].telo.$

$[//].[=].[id].[;].program$

$proc \rightarrow [void].[id].parametr.[=].$

$telo.[//].program$

$parametr \rightarrow typ.[:].[id].parametr$

$parametr \rightarrow \epsilon$

$fun \rightarrow typ.[id].parametr.[=].telo.$

$[//].[=].[id].[;].program$

$program \rightarrow \epsilon$

$typ \rightarrow [int]$

$typ \rightarrow [string]$

$telo \rightarrow lokdec.prikaz$

$lokdec \rightarrow [<<].dcl.dcl\_list.[>>]$

$dcl\_list \rightarrow [;].dcl.dcl\_list$

$dcl\_list \rightarrow \epsilon$

$lokdec \rightarrow \epsilon$

$prikaz \rightarrow [if].vyraz.[].prikaz.[].$

$else.prikaz$

$else \rightarrow [].prikaz.[]$

$else \rightarrow \epsilon$

$prikaz \rightarrow [do].prikaz.$

$[(].vyraz.[)].prikaz$

$prikaz \rightarrow [id].test.prikaz$

$test \rightarrow [(].call\_param.[)]$

$test \rightarrow [=].vyraz.[;]$

$call\_param \rightarrow vyraz.vyraz\_list$

$call\_param \rightarrow \epsilon$

$vyraz\_list \rightarrow [,].vyraz.vyraz\_list$

$vyraz\_list \rightarrow \epsilon$

$prikaz \rightarrow \epsilon$

$vyraz \rightarrow or\_op.ro$

$ro \rightarrow [or].or\_op.ro$

$ro \rightarrow \epsilon$

$or\_op \rightarrow and\_op.dna$

$dna \rightarrow [and].and\_op.dna$

$dna \rightarrow \epsilon$

$and\_op \rightarrow [not].and\_op$

$and\_op \rightarrow rovno\_op.onvor$

$onvor \rightarrow [==].rovno\_op.onvor$

$onvor \rightarrow [<>].rovno\_op.onvor$

$onvor \rightarrow \epsilon$

$rovno\_op \rightarrow menvet\_op.tevnem$

$tevnem \rightarrow [>].menvet\_op.tevnem$

$tevnem \rightarrow [>=].menvet\_op.tevnem$

$tevnem \rightarrow [<].menvet\_op.tevnem$

$tevnem \rightarrow [<=].menvet\_op.tevnem$

$tevnem \rightarrow \epsilon$

$menvet\_op \rightarrow plmi\_op.imlp$

$imlp \rightarrow [+].plmi\_op.imlp$

$imlp \rightarrow [-].plmi\_op.imlp$

$imlp \rightarrow \epsilon$

$plmi\_op \rightarrow krde\_op.edrk$

$edrk \rightarrow [*].krde\_op.edrk$

$edrk \rightarrow [div].krde\_op.edrk$

$edrk \rightarrow [mod].krde\_op.edrk$

$edrk \rightarrow \epsilon$

$krde\_op \rightarrow [-].krde\_op$

$krde\_op \rightarrow [id].test2$

$test2 \rightarrow [(].call\_param.[)]$

$test2 \rightarrow \epsilon$

$krde\_op \rightarrow [(].vyraz.[)]$

$krde\_op \rightarrow [strExp]$

$krde\_op \rightarrow [intExp]$

3 Beyond classic LL parsing

The main goal of regulated formal systems is to extend abilities from standard CF LL-parsing to CS or RE families with preservation of ease of parsing.

In [Kol–04] and [Rych–05] we can find some basic facts from theory of regulated pushdown automata. We figured that regulated pushdown automata can in some cases simulate Turing machines so we could use this theory for constructing parsers for context-sensitive languages or even type-0 languages.

We have also demonstrated the basic problem of this concept: complexity. Almost trivial Turing machine was transformed to regulated pushdown automata with almost 6 000 rules.

3.1 Parsing context-sensitive languages

As we have seen in previous lines, converting D(LB)TM or SCG to DRPA is very complex task. For the most simple context-sensitive languages corresponding DRPA has thousands of rules. If we want to use these algorithms for creating some practical parser for real context-sensitive programming language it may result in millions of rules. Therefore, we are looking for another way to parse context-sensitive languages. We would like to extend some context-free grammar of any common programming language (such as Pascal, C/C# or Java). After extending context-free grammar to corresponding context-sensitive grammar, parsing should be straightforward.

As an example of a context-free language we use a language called ZAP03 [ZAP–03] which has very similar syntax to Pascal. The ZAP03 grammar is described in Appendix C. We need the following fragment of the ZAP03 grammar which describes variable definition:
`DCL` $\rightarrow$ `TYP [:]  [id] ID_LIST`
`ID_LIST` $\rightarrow$ `[,] [id] ID_LIST`
`ID_LIST` $\rightarrow \epsilon$
and fragment of assignment statement
`CMD` $\rightarrow$ `[id] TEST CMD`
`TEST` $\rightarrow$ `[=] STMT [;]`
and using variable in command
`OPER` $\rightarrow$ `[id] TEST2`
`TEST2` $\rightarrow \epsilon$.
Symbols in brackets [,] are terminals. Complete ZAP03 grammar has about 70 context-free grammar rules.

We denote the context-sensitive extension of ZAP03 language as KontextZAP03. We define grammar of language KontextZAP03 in the following way. Substitute previous rules with these scattered-context ones:
`(DCL, S')` $\rightarrow$ `(TYP [:]  [id] ID_LIST, D)`
`(ID_LIST, S')` $\rightarrow$ `([,] [id] ID_LIST, D)`
`(ID_LIST)` $\rightarrow$ (ϵ)
assignment statement

`(CMD, D)` $\rightarrow$ `([id] TEST CMD, DL)`
`(TEST)` $\rightarrow$ `([=] STMT [;])`
and using variable in command
`(OPER, D)` $\rightarrow$ `([id] TEST2, DR)`
`(TEST2)` $\rightarrow$ (ϵ).

Parsing now proceeds in the following way: starting symbol is `S S'` and derivation will go as usual until there is `DCL S'` processed. At this moment `S'` is rewritten to `D` indicating that variable [id] is defined. When variable [id] is used on left resp. right side of assignment D is rewritten to DL resp. DR according to second resp. third previously shown fragment. If variable [id] is used without being defined beforehand, a parse error occurs because `S'` is not rewritten to `D` and `S'` cannot be rewritten to `DL` or `DR`. When the input is parsed `S` is rewritten to epsilon and only `D`$\{$`LR`$\}^*$ remains.

$$S\ S' \Rightarrow^* DCL\ S' \Rightarrow TYP\ [:]\ [id]\ ID\_LIST\ D \Rightarrow^* CMD\ D \Rightarrow$$
$$\Rightarrow [id]\ TEST\ CMD\ DL \Rightarrow^* DL$$

If the only remaining symbol is D, it means that variable [id] was defined but never used. If `DL`$^+$ is the only remaining symbol, we know that variable [id] was defined and used only on the left sides of assignments. Finally if there is the only remaining `DR(LR)`*, we know that the first occurrence of variable [id] is on the right side of an assignment statement and therefore it is being read without being set. In all these cases the compiler should generate a warning. These and similar problems are usually addressed by a data-flow analysis phase carried out during semantic analysis.

Using this algorithm we can only process one variable at a time. But the proposed mechanism can be easily extended to a finite number of variables by adding new `S'`$\cdots$`'` every time we discover a variable definition.

Because original ZAP03 grammar is LL1 and using described algorithm wasn't any rule added, KontextZAP03 grammar has unambiguous derivations

Parsing of described SC grammar can be implemented by pushdown automaton with finite number of pushdowns.

Example program in ZAP03 language can be

```
int :  a, b, c, d;
string :  s;

begin
  a = 1;
  b = 2;
  c = 10;
  d = 15;
  s = "foo";
  a = c;
end
```

Another program is well formed according to ZAP03 grammar, but it's semantics is not correct.

```
int :  a, b, c, d;
string :  s;

begin
  a = 1;
  b = 2;
  d = 15;
  s = "foo";
  a = c;
end
```

Corresponding stack to variable `C` will be `DR` what lead to warning:

```
Variable c read but not set.
```

The last example shows another CS extension

```
int :  a, b, c, d;
string :  s;

begin
  a = 1;
  d = 15;
  s = "foo";
  a = c;
end
```

Corresponding stack to variable `B` will be `D` what lead to warning:

```
Variable c read but not set.
Variable b is defined but never used.
```

3.2 Type checking

By using Scattered-Context grammars we can describe type set of language directly in grammar. Almost trivial language with type check using 5 stacks can look like this:

$(s) \rightarrow (program)$
$(program) \rightarrow (dcl)$
$(program) \rightarrow ([begin]prikaz[end])$
$(dcl) \rightarrow (typ[:]dcl2)$
$(dcl2, S, INT, ,) \rightarrow$
$\quad ([id]id\_list[;]next, DCL, INT, INT)$
$(dcl2, S, STR, ,) \rightarrow$
$\quad ([id]id\_list[;]next, DCL, STR, STR)$
$(id\_list) \rightarrow ([,]id_{l}ist2)$
$(id\_list2, S) \rightarrow ([id]id\_list, DCL)$
$(id\_list) \rightarrow ()$
$(next) \rightarrow (dcl)$
$(next) \rightarrow (program)$
$(typ, ,) \rightarrow ([int], , INT)$
$(typ, ,) \rightarrow ([string], , STR)$
$(prikaz, DCL, INT, ,) \rightarrow$
$\quad ([id]testprikaz, DCLUSDL, INT, INT)$
$(prikaz, DCL, STR, ,) \rightarrow$
$\quad ([id]testprikaz, DCLUSDL, STR, STR)$
$(prikaz) \rightarrow ()$

$(test) \rightarrow ([=]vyraz[;])$ $(vyraz,,,,INT) \rightarrow ([digit],,,,)$
$(vyraz, DCL) \rightarrow ([id], DCLUSDR)$ $(vyraz,,,,STR) \rightarrow ([strval],,,,)$

3.3 Other applications of CS languages

It is obvious, that using a simple Scattered Context extension of CF languages, we obtain a grammar with interesting properties with respect to analysis of a programming language source code. We provide some basic motivation examples.

1. Errors related to usage of undefined variables may be discovered and handled at parse time without the need to handle them by static semantic analysis.

2. A CS extension of a grammar of the Java programming language, which copes with problems like mutual exclusion of various keywords, such as `abstract` and `final`, reflecting the fact that abstract methods cannot be declared final and vice versa. This situation can be handled quite easily, by introducing additional symbol S'' and two corresponding rules $S'' \rightarrow A$ (corresponds to `abstract`) and $S'' \rightarrow F$ (corresponds to `final`). Obviously only one of the rules can be used at a time.

3. Introducing an `observer` keyword for methods in Java, which indicates that this method does not modify the state of *this* object (similar to defining method as `const` in C++). Handling of such keyword in the language grammar is similar to approach taken in the previous example.

4. Accounting of statements in a program in a ZAP03 language by introducing new `size` keyword, which defines upper bound on the number of statements in current scope. The parser is than extended such that when the keyword is discovered, the number of specialised nonterminals (say `X`) is generated on the stack – as specified by the keyword occurrence and the grammar of the language is modified accordingly.
   The rule:
   `PRIKAZ → [id] TEST PRIKAZ`
   changes to:
   `(PRIKAZ, X) → ([id] TEST PRIKAZ,` ϵ `)`
   Then, when a statement rule is used, one `X` nonterminal is eliminated from the stack. If there are no remaining `X` nonterminals, the parsing immediately fails. The language used in this example is:
   $$L(G) = \{w.|w|_{10}\},$$
   where w is word and $|w|_{10}$ is the length of w written as a decadic number.

3.4 Formal Program Verification

One of the promising applications of the described concept is introducing a notion of *preconditions* and *postconditions* to the ZAP03 programming language. Preconditions and postconditions are essentially sets of logical formulae, which are required to hold at entering resp. leaving a program or method. For example, when computing a square root of x, we can require a positivity of x using precondition $\{x >= 0\}$. A computation of sinus function $\{y = sinx\}$ a natural postcondition $\{(y <= 1) \wedge (y >= -1)\}$ arises.

Statements of a programming languages then induce transformation rules on precondition and postcondition sets. For example, assignment statement in the form $V := E$ defines a transformation:

$$\{P[E/V]\}V := E\{P\},$$

where V is a variable, E is an expression, P is precondition and P[E/V] denotes a substitution of V for all occurences of E in P. Other transformation examples may be found in [Gor–98].

The purpose of introducing preconditions and postconditions into a language is to be able to derive postconditions from specified preconditions using transformation rules in a particular program. Such program than carries a formal proof of its correctness with it, which is a desirable property.

In the ZAP03 language we can implement the described concept by introducing two new keywords `pre` and `post` and by extending the rules of the language grammar with abovementioned transformation rules. When the parsing of a program is initiated, the precondition set is constructed using the `pre` declarations and the transformation rules are applied to it as the parsing progresses through the source code. When the parsing terminates, the resulting set of transformed preconditions is compared with the declared postconditions. If these two sets match, the parsed program is correct with respect to the specified preconditions and postconditions.

However, for practical reasons we must define constraints on the possible preconditions and postconditions. Postconditions must be generally derivable from preconditions or (as a corollary of the Godels incompleteness theroem) the postconditions need not be provable from the preconditions at all, but may still hold. In this particular case we have encountered a program which may be correct, but we cannot verify this fact.

Conclusions

In this work presented some ideas from parsing Context-Sensitive languages. They are very powerful extension of classic parsing techniques. They let us discover some common errors such as undefined or unused variables in parsing time. The extension suggested in this work is far more beyond. In some conditions we are able to add some features to language that enables us to check formal program verification.

References

[Gor–98] Gordon, J. C. M.: Programming Language Theory and its Implementation, 1998.

[Kol–04] Kolář, D.: Pushdown Automata: New Modifications and Transformations, Habilitation Thesis, 2004.

[Med–00] Meduna, A., Kolář, D.: Reguleted Pushdown Automata, Acta Cybernetica, Vol. 14, 2000. Pages 653-664.

[Med–03] Meduna, A., Fernau, H.: On the degree of scattered context-sensitivity, Elsevier, Theoretical Computer Science 290 (2003), Pages 2121-2124.

[Sal–96a] Salomaa, A.: Handbook of Formal Languages vol. 1

[Sal–96b] Salomaa, A.: Handbook of Formal Languages vol. 2

[Sal–73] Salomaa, A.: Formal Languages, Academic Press, New York, 1973.

[ZAP–03] http://www.fit.vutbr.cz/study/courses/ZAP/public/project/

[Kol–04] Kolar, D.: Pushdown Automata: New Modifications and Transformations, Habilitation Thesis.

[Med–00] Meduna, A., Kolar, D.: Reguleted Pushdown Automata, Acta Cybernetica, Vol. 14, 2000. Pages 653-664.

[Roz–80] Rozenberg, G., Engelfriet, J.: Fixed Point Languages, Equality Languages, and Representation of Recursively Enumerable Languages, Journal of ATM, Vol. 27, 1980. Pages 499-518.

[Sal–73] Salomaa, A.: Formal Languages, Academic Press, New York, 1973.

[Uni–05] http://www.unidex.com/turing/tmml.dtd

[Ber–79] Berstel, J.: Transductions and Context-Free Languages, G. B. Teubner, Stuttgart 1979.

[Met–01] Metsker, S.: Building Parsers With Java, Addison-Wesley 2001.

[Lis–00] Liskov, B.: Program Development in Java, Addison-Wesley 2000.

[Aho–72] Aho, A., Ullman, J.: The Theory of Parsing, Translation, and Compiling, Prentice-Hall INC. 1972.

[Rych–05] Rychnovsky, L.: Vztah řízených zásobníkových automatů a Turingových strojů, Report from semestral project, 2005.