Extensible LL(1) parser

Michal Žemlička*

November 14, 1996

Abstract

In this poster is shown why and how to construct extensible parsers for LL(1) languages. There are also introduced kind grammars as new form for defining LL(1) languages.

1 Motivation

Programmers usually want to write short and easy-to-read programs. Several times repeating statement sequences are closed into procedures and instead of writing this sequences only the name of procedure is written. Some programming languages allows also using of macros. Sometimes we also need to add new semantics to existing syntactic constructions and/or define new ones.

Having extensible languages and compilers we would be able to write programs like this:

```
program powering;
function power(base,exponent);
  var result;
  begin
    result:=1;
    while base>0 do result:=result*base; exponent:=exponent-1 end;
    power:=result
  end;
terminal 'doublestar','**';
rule 'power:<F>:=<f><doublestar><F>';
instance of 'power' is power;
var base, exponent;
begin
  write 'Enter base and exponent:  '; read base; read exponent;
  write 'Power is '; write base**exponent
end.
```

Expression was originally defined so:

```
<E>::=<E>(plus)<T>|<E>(minus)<T>|<T>
<T>::=<T>(star)<F>|<F>
<F>::=<f>
<f>::={num}|{id}
```

For construction of extensible compilers we need extensible parsers and that is what we will write about.

2 Kind Grammars

Definition (Kind grammar)
Class of kind grammars KG contains the grammars fulfilling the following conditions:

1. $LRF_G(A) \cap NLRF_G(A) = \{\}$ $\quad \forall A \in N_G$
2. $A \to \alpha\beta, A \to \alpha\gamma \in P_G \implies First(\beta) \cap First(\gamma) = \{\} \ \vee \ First(\beta) = First(\gamma)$
3. $A \to \alpha\beta, A \to \alpha \in P_G \ \wedge \ \beta \neq \varepsilon \implies First(\beta) \cap Follow(A) = \{\}$

[1] Department of Software Engineering, Charles University, Prague; zemlicka@kki.ms.mff.cuni.cz

where $LRP_G = \{A \to A\alpha | A \to A\alpha \in P_G\}$ stays for left recusive productions, $NLRP_G = P_G - LRP_G$ for productions without left recursion,

$$LRF_G(A) = \bigcup_{A \to A\alpha \in LRP_G} First(\alpha)$$

for left recursive $Follow$, and

$$NLRF_G(A) = \bigcup_{B \to \alpha\, A\beta \in NLRP_G} First(\beta)$$

for $Follow$ without left recursion.

If we omit productions with left recursion and set α in second condition to ε (empty string), we get definition of LL(1) grammars. We can treat LL(1) grammars as special class of kind grammars.

There is an algorithm (left factorization) translating kind grammar to LL(1) grammar.

3 Kind automata

Definition ($=_G$).
Let $G \in KG$. Relation $=_G$:

$$A \to \alpha.\beta \;=_G\; B \to \gamma.\delta \quad \equiv_{df} \quad (A = B) \wedge (\alpha = \gamma)$$

defined on items from G is reflexive, symmetric, transitive – so it is equivalence. Equivalence classes by $=_G$ will be $[\ldots]_{=_G}$.

Kind automaton is constructed as follows:

Construction (Kind automata).

Let $G = (N_G, T_G, S_G, P_G)$, $M = (Q_M, T_M, K_M, \delta_M, q_0, Z_0, F_M)$. Then:

$Q_M = \{[A \to \alpha.\beta]_{=_G} | A \to \alpha\beta \in P_G\}$, $T_M = T_G$, $K_M = Q_M$,
$\delta_M = \{([A \to \alpha.a\beta]_{=_G}, a\sigma, X) \to ([A \to \alpha a.\beta]_{=_G}, \sigma, X) | A \to \alpha a\beta \in P_G\}$
$\cup \{([A \to \alpha.B\beta]_{=_G}, \sigma, X) \to ([B \to .\gamma]_{=_G}, \sigma, [A \to \alpha.B\beta]_{=_G} X)$
$\quad | A \to \alpha B\beta, B \to \gamma \in P_G\}$
$\cup \{([B \to .]_{=_G}, a\sigma, [A \to \alpha.B\beta]_{=_G} X) \to ([A \to \alpha B.\beta]_{=_G}, a\sigma, X)$
$\quad | A \to \alpha B\beta \in P_G, a \in NLRF_G(B)\}$
$\cup \{([A \to \alpha.]_{=_G}, a\sigma, X) \to ([A \to A.\beta]_{=_G}, a\sigma, X) | a \in LRF_G(A)\}$

$q_0 = [S_G \to \alpha]_{=_G}$,
$Z_0 = \{\}$,
$F_M = \{[S_G \to \alpha.]_{=_G} | S_G \to \alpha \in P_G\}$.

Adding new production to kind automaton is very easy: just add new terminals to terminal set, new item classes to state set and stack alphabet, and add new deductions to the δ-function.

4 Conclusion

It was shown, how to construct an extensible parser for LL(1) languages. In addition, the definition grammar is not so strict as obvious LL(1) grammar.

Extensible compiler for LL(1) languages was written as my Master Thesis [?]. This compiler can also translate more than one language.

Extensible parsers/compilers based on buLC algorithm and LR-techniques are under development.

References

[Aho86] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullmann: Compilers Principles, Techniques, and Tools, Addison Wesley, 1986

[Zem94] Michal Žemlička: Compiler for Extensible LL(1)-language Based on Pascal Language (in Czech), MFF UK, 1994