Compiler Construction

Lecture 8: Syntactic Analysis IV (Practical Issues in LL Parsing)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/cc08/

Summer semester 2008

Outline

Lookahead Sets

Definition (Lookahead set)

Given $\pi = A \to \beta \in P$,

$$\mathrm{la}(\pi) := \mathrm{fi}(\beta \cdot \mathrm{fo}(A)) \subseteq \Sigma_\varepsilon$$

is called the lookahead set of π (where $\mathrm{fi}(\Gamma) := \bigcup_{\gamma \in \Gamma} \mathrm{fi}(\gamma)$).

Corollary

1. *For all* $a \in \Sigma$,
$$a \in \mathrm{la}(A \to \beta) \text{ iff } a \in \mathrm{fi}(\beta) \text{ or } (\beta \Rightarrow^* \varepsilon \text{ and } a \in \mathrm{fo}(A))$$
2. $\varepsilon \in \mathrm{la}(A \to \beta)$ *iff* $\beta \Rightarrow^* \varepsilon$ *and* $\varepsilon \in \mathrm{fo}(A)$

Characterization of $LL(1)$

Theorem (Characterization of $LL(1)$)

$G \in LL(1)$ iff for all pairs of rules $A \to \beta \mid \gamma \in P$ (where $\beta \neq \gamma$):

$$\mathrm{la}(A \to \beta) \cap \mathrm{la}(A \to \gamma) = \emptyset.$$

Proof.

on the board □

Remark: the above theorem generally does not hold if $k > 1$
(cf. exercises)

Deterministic Top-Down Parsing

Approach: given $G \in CFG_\Sigma$,

1. Verify that $G \in LL(1)$ by computing the lookahead sets and checking alternatives for disjointness
2. Start with nondeterministic top-down parsing automaton NTA(G)
3. Use 1-symbol lookahead to control the choice of expanding productions:
   - $(aw, A\alpha, z) \vdash (aw, \beta\alpha, zi)$ if $\pi(i) = A \to \beta$ and $a \in \mathrm{la}(\pi(i))$
   - $(\varepsilon, A\alpha, z) \vdash (\varepsilon, \beta\alpha, zi)$ if $\pi(i) = A \to \beta$ and $\varepsilon \in \mathrm{la}(\pi(i))$
   - [as before: $(aw, a\alpha, z) \vdash (w, \alpha, z)$]

$\Longrightarrow$ deterministic top-down parsing automaton DTA(G)

Remarks:

- DTA(G) is actually not a pushdown automaton (a is read but not consumed). But: can be simulated using the finite control.
- Advantage of using lookahead is twofold:
  - Removal of nondeterminism
  - Earlier detection of syntax errors (in configurations $(aw, A\alpha, z)$ where $a \notin \bigcup_{A \to \beta \in P} \mathrm{la}(A \to \beta)$)

Outline

Transformation to $LL(1)$

Assume that $G = \langle N, \Sigma, P, S \rangle \in CFG_\Sigma \setminus LL(1)$
(i.e., there exist $A \to \beta \mid \gamma \in P$ such that $\mathrm{la}(A \to \beta) \cap \mathrm{la}(A \to \gamma) \neq \emptyset$)

Two heuristics for transforming G into $G' \in LL(1)$:

1. Removal of left recursion
2. Left factorization

(used in parser-generating systems such as ANTLR)

Remarks:

- Transformations generally preserve the semantics (= generated language) of CFGs but not the syntactic structure of words (different syntax trees).
- Transformations cannot always yield an $LL(1)$ grammar (since not every context-free language is generated by an LL grammar; details later).

Left Recursion I

Definition 8.1 (Left recursion)

A grammar $G = \langle N, \Sigma, P, S \rangle \in CFG_\Sigma$ is called left recursive if there exist $A \in N$ and $\alpha \in X^*$ such that $A \Rightarrow^+ A\alpha$.

Corollary 8.2

If $G \in CFG_\Sigma$ is left recursive with $A \Rightarrow^+ A\alpha$, then there exists $\beta \in X^*$ such that $A \Rightarrow_l^+ A\beta$.

Example 8.3

The grammar (cf. Example 5.11)

$$G_{AE}: \quad \begin{aligned} E &\to E\texttt{+}T \mid T \\ T &\to T\texttt{*}F \mid F \\ F &\to \texttt{(}E\texttt{)} \mid \texttt{a} \mid \texttt{b} \end{aligned}$$

is left recursive, and in Example 7.8 it was shown that $G_{AE} \notin LL(1)$

Left Recursion II

Lemma 8.4

If $G \in CFG_\Sigma$ is left recursive, then $G \notin \bigcup_{k\in\mathbb{N}} LL(k)$.

Proof.

(for $k = 1$) Assume that $G \in LL(1)$ is left recursive with $A \Rightarrow_l^+ A\beta$.
Together with the reducedness of G this implies that
$S \Rightarrow_l^* vA\alpha \Rightarrow_l^+ vA\beta\alpha \Rightarrow_l^+ vw$ for some $v, w \in \Sigma^*$ and $\alpha \in X^*$.
The corresponding computation of DTA(G) (Def. 7.9) starts with
$(vw, S, \varepsilon) \vdash^* (w, A\alpha, \ldots) \vdash^+ (w, A\beta\alpha, \ldots)$.
But in the last state the behaviour of DTA(G) is determined by the same input (fi(w)) and stack symbol (A). Thus it enters a loop of the form $(w, A\alpha, \ldots) \vdash^+ (w, A\beta\alpha, \ldots) \vdash^+ (w, A\beta\beta\alpha, \ldots) \vdash^+ \ldots$ and will never recognize w. Contradiction □

Removing Direct Left Recursion

Direct left recursion occurs in productions of the form

$$A \to A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n \qquad \text{where } \alpha_i \neq \varepsilon \text{ and } \beta_j \neq A\ldots$$

Transformation: replacement by right recursion

$$\begin{aligned} A &\to \beta_1 A' \mid \ldots \mid \beta_n A' \\ A' &\to \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon \end{aligned}$$

(with a new $A' \in N$) which preserves $L(G)$.

Example 8.5

G_{AE} :

$$\begin{aligned} E &\to E\texttt{+}T \mid T \\ T &\to T\texttt{*}F \mid F \\ F &\to \texttt{(}E\texttt{)} \mid \texttt{a} \mid \texttt{b} \end{aligned}$$

is transformed into

G'_{AE} :

$$\begin{aligned} E &\to TE' \\ E' &\to \texttt{+}TE' \mid \varepsilon \\ T &\to FT' \\ T' &\to \texttt{*}FT' \mid \varepsilon \\ F &\to \texttt{(}E\texttt{)} \mid \texttt{a} \mid \texttt{b} \end{aligned}$$

with $G'_{AE} \in LL(1)$ (see Example 7.8).

Removing Indirect Left Recursion

Indirect left recursion occurs in productions of the form ($n \geq 1$)

$$\begin{array}{ll} A & \rightarrow A_1\alpha_1 \mid \ldots \\ A_1 & \rightarrow A_2\alpha_2 \mid \ldots \\ & \vdots \\ A_{n-1} & \rightarrow A_n\alpha_n \mid \ldots \\ A_n & \rightarrow A\beta \mid \ldots \end{array}$$

Transformation: into Greibach Normal Form with productions of the form

$$A \rightarrow aB_1 \ldots B_n \qquad \text{(where } B_i \neq S\text{) or}$$
$$S \rightarrow \varepsilon$$

(cf. *Automata Theory and Formal Languages*)

Left Factorization

Applies to productions of the form

$$A \to \alpha\beta \mid \alpha\gamma$$

which are problematic if α "longer than" lookahead.

Transformation: delaying the decision by left factorization

$$\begin{aligned} A &\to \alpha A' \\ A' &\to \beta \mid \gamma \end{aligned}$$

(with a new $A' \in N$) which preserves $L(G)$.

Example 8.6

$$\begin{aligned} \mathit{Statement} \to\ & \mathsf{if}\ \mathit{Condition}\ \mathsf{then}\ \mathit{Statement}\ \mathsf{else}\ \mathit{Statement}\ \mathsf{fi} \\ \mid\ & \mathsf{if}\ \mathit{Condition}\ \mathsf{then}\ \mathit{Statement}\ \mathsf{fi} \end{aligned}$$

is transformed into

$$\begin{aligned} \mathit{Statement} &\to \mathsf{if}\ \mathit{Condition}\ \mathsf{then}\ \mathit{Statement}\ S' \\ S' &\to \mathsf{else}\ \mathit{Statement}\ \mathsf{fi} \mid \mathsf{fi} \end{aligned}$$

Outline

The Complexity of $LL(1)$ Parsing I

- $LL(1)$ parsing has time (and hence space) complexity $\mathcal{O}(|w|)$ (where $w \in \Sigma^*$ is the input word)
- Here: proof for ε-free grammars (i.e., $A \to \alpha \in P \implies \alpha \neq \varepsilon$)
- General case: see O. Mayer: *Syntaxanalyse*, p. 211ff

Lemma 8.7

Let $G = \langle N, \Sigma, P, S \rangle \in LL(1)$ *be* ε*-free. If*

$$(w, S, \varepsilon) \vdash^n (\varepsilon, \varepsilon, z)$$

in $\mathrm{DTA}(G)$*, then*

$$n \leq |w| \cdot |N| + 1.$$

Proof.

Let $(w, S, \varepsilon) \vdash^n (\varepsilon, \varepsilon, z)$ in $\mathrm{DTA}(G)$. To show: $n \leq |w| \cdot |N| + 1$

1. Clear: the computation involves $|w|$ matching steps and one accept-step.
2. Since G is ε-free, every matching step is preceded by k expansion steps of the form
$$\begin{aligned}(av, A_1\alpha_1, \ldots) &\vdash (av, A_2\alpha_2\alpha_1, \ldots)\\ &\vdots\\ &\vdash (av, A_k\alpha_k \ldots \alpha_1, \ldots)\\ &\vdash (av, a\alpha_{k+1} \ldots \alpha_1, \ldots)\end{aligned}$$
where $A_i \to A_{i+1}\alpha_{i+1}$ for each $i \in [k-1]$ and $A_k \to a\alpha_{k+1}$.
3. This implies that $A_i \neq A_j$ for $i \neq j$ (by Lemma 8.4, G is not left recursive), and hence $k \leq |N|$.
4. Altogether: $n \leq |w| \cdot |N| + 1$.

□

Outline

Recursive-Descent Parsing I

Idea: avoid explicit use of pushdown store (as in DTA(G)) by employing recursive procedures (with implicit runtime stack)

Advantage: simple implementation

Ingredients:

- variable `token` for current token
- function `next()` for invoking the scanner
- procedure `print(i)` for displaying the leftmost analysis (or errors)

Method: to every $A \in N$ we assign a procedure `A()` which

- tests `token` with regard to the lookahead sets of the A-productions,
- prints the corresponding rule number and
- evaluates the corresponding right-hand side as follows:
  - for $a \in \Sigma$: check `token`; call `next()`
  - for $A \in N$: call `A()`

Recursive-Descent Parsing II

Example 8.8 (Arithmetic expressions; cf. Example 8.5)

```
proc main();
  token := next(); E()
proc E();    (* E → T E' *)
  if token in {'(','a','b'} then print(1); T(); E'()
  else print(error); stop fi
proc E'();   (* E' → + T E' | ε *)
  if token = '+' then print(2); token := next(); T(); E'()
  elsif token in {EOF, ')'} then print(3)
  else print(error); stop fi
proc T();    (* T → F T' *)
  if token in {'(','a','b'} then print(4); F(); T'()
  else print(error); stop fi
proc T'();   (* T' → * F T' | ε *)
  if token = '*' then print(5); token := next(); F(); T'()
  elsif token in {'+',EOF,')'} then print(6)
  else print(error); stop fi
proc F();    (* F → ( E ) | a | b *)
  if token = '(' then print(7); token := next(); E();
    if token = ')' then token := next() else print(error); stop fi
  elsif token = 'a' then print(8); token := next()
  elsif token = 'b' then print(9); token := next()
  else print(error); stop fi
```

Outline

Error Handling

Error configurations of DTA(G):

- $(aw, A\alpha, z)$ where $a \notin \bigcup_{A \to \beta \in P} \mathrm{la}(A \to \beta)$ $(\Longrightarrow \mathrm{act}(a, A) = \mathsf{error})$
- $(\varepsilon, A\alpha, z)$ where $\varepsilon \notin \bigcup_{A \to \beta \in P} \mathrm{la}(A \to \beta)$ $(\Longrightarrow \mathrm{act}(\varepsilon, A) = \mathsf{error})$
- $(aw, b\alpha, z)$ where $a \neq b$ $(\Longrightarrow \mathrm{act}(a, b) = \mathsf{error})$
- $(\varepsilon, b\alpha, z)$ $(\Longrightarrow \mathrm{act}(\varepsilon, b) = \mathsf{error})$
- (aw, ε, z) $(\Longrightarrow \mathrm{act}(a, \varepsilon) = \mathsf{error})$

Observation: correct prefix property of LL parsing, i.e., syntactic errors are detected at the earliest possible position (every input prefix which does not produce an error can be extended to a word $w \in L(G)$)

Does not mean: error is recognized at the position where it is caused!

Example: assignment `a := b * c - (d + e));`
Possible corrections:

- remove closing bracket: `a := b * c - (d + e);`
- insert opening bracket: `a := b * (c - (d + e));`

The General Problem

- Let $w = xy \in \Sigma^*$ be the input word such that x is the longest prefix of a word in $L(G)$ (i.e., the error is detected at the first symbol of y) and $w \notin L(G)$.
- Parser makes assumption about error type and corrects w accordingly:
  - Assumes prefix x' of x to be correct
  - Correct prefix property
    $\Longrightarrow$ there exists $z \in \Sigma^*$ such that $x'z \in L(G)$
  - Parser chooses prefix z' of z and suffix y' of y
  - Parsing resumed with input $w' := x'z'y'$ (at first symbol of z') (error recovery)
- Desirable properties of correction:
  - At least one symbol of y' can be processed before next error occurs (if $y' \neq \varepsilon$)
  - Preserve as many symbols of w as possible (i.e., x' and y' "long" and z' "short")
- $x' \neq x$ hard to implement, therefore usually $x' := x$

Aspects of Error Handling

Further criteria for "good" error handling:

- Continuation of parsing in any case, independent of severity of error
- Frequency of correct error diagnosis
- Suppression of subsequent errors
- Complexity of analyzing correct inputs not impaired

Observation: no "best method" available

- correction not unique
- experience of programmer
- peculiarities of (programming) language

$\Longrightarrow$ employ heuristics

Panic Mode

Simplest form of error handling: panic mode
Upon occurrence of an error,

- skip input symbols
- until a token in a selected set of "separating" or "closing" tokens appears (synchronizing tokens)

Example: suitable synchronizing tokens in imperative languages for

- assignments: ";"
- declarations: ";" or ","
- control structures: fi or od
- blocks: end

Challenge: choose set of synchronizing tokens such that

- parser recovers quickly from errors that are likely to occur and
- not too much input is overread

(see Aho/Lam/Sethi/Ullman: *Compilers: Principles, Techniques, and Tools*, 2nd ed., p. 228ff)

Error Handling Example I

Example 8.9 (cf. Example 7.10)

$$
\begin{array}{lll}
G'_{AE}: & E \to TE' & (1) \\
& E' \to \texttt{+}TE' \mid \varepsilon & (2,3) \\
& T \to FT' & (4) \\
& T' \to \texttt{*}FT' \mid \varepsilon & (5,6) \\
& F \to \texttt{(}E\texttt{)} \mid \texttt{a} \mid \texttt{b} & (7,8,9)
\end{array}
$$

$A \in N$	fo(A)
E	$\{\varepsilon, \texttt{)}\}$
E'	$\{\varepsilon, \texttt{)}\}$
T	$\{\texttt{+}, \varepsilon, \texttt{)}\}$
T'	$\{\texttt{+}, \varepsilon, \texttt{)}\}$
F	$\{\texttt{*}, \texttt{+}, \varepsilon, \texttt{)}\}$

With synchronizing tokens from fo sets:

act : $\Sigma_\varepsilon \times X_\varepsilon \to \{(\alpha, i) \mid \pi(i) = A \to \alpha\} \cup \{\mathsf{pop}, \mathsf{accept}, \mathsf{error}, \mathsf{sync}\}$ (empty = error)

act	E	E'	T	T'	F	a	b	(	)	*	+	ε
a	$(TE', 1)$		$(FT', 4)$		(a, 8)	pop	sync	sync	sync	sync	sync	
b	$(TE', 1)$		$(FT', 4)$		(b, 9)	sync	pop	sync	sync	sync	sync	
(	$(TE', 1)$		$(FT', 4)$		$((E), 7)$	sync	sync	pop	sync	sync	sync	
)	sync	$(\varepsilon, 3)$	sync	$(\varepsilon, 6)$	sync	sync	sync	sync	pop	sync	sync	
*				$(*FT', 5)$	sync	sync	sync	sync	sync	pop	sync	
+		$(+TE', 2)$	sync	$(\varepsilon, 6)$	sync	sync	sync	sync	sync	sync	pop	
ε	sync	$(\varepsilon, 3)$	sync	$(\varepsilon, 6)$	sync	sync	sync	sync	sync	sync	sync	accept

Error Handling Example II

Example 8.9 (continued)

act	E	E'	T	T'	F	a	b	(	)	*	+	ε
a	$(TE', 1)$		$(FT', 4)$		$(\mathtt{a}, 8)$	pop	sync	sync	sync	sync	sync	
b	$(TE', 1)$		$(FT', 4)$		$(\mathtt{b}, 9)$	sync	pop	sync	sync	sync	sync	
(	$(TE', 1)$		$(FT', 4)$		$((E), 7)$	sync	sync	pop	sync	sync	sync	
)	sync	$(\varepsilon, 3)$	sync	$(\varepsilon, 6)$	sync	sync	sync	sync	pop	sync	sync	
*				$(*FT', 5)$	sync	sync	sync	sync	sync	pop	sync	
+		$(+TE', 2)$	sync	$(\varepsilon, 6)$	sync	sync	sync	sync	sync	sync	pop	
ε	sync	$(\varepsilon, 3)$	sync	$(\varepsilon, 6)$	sync	sync	sync	sync	sync	sync	sync	accept

Meaning of table entries:

- $\text{act}(x, Y) = \text{sync}$
  $\Longrightarrow$ pop Y and resume parsing
- $\text{act}(x, A) = \text{error}$
  $\Longrightarrow$ skip x and resume parsing

$$
\begin{array}{ll}
 & (\mathtt{+a*+b},\ E\ ,\ \varepsilon\ ) \\
\vdash & (\ \mathtt{a*+b},\ E\ ,\ \varepsilon\ ) \\
\vdash & (\ \mathtt{a*+b},\ TE'\ ,\ 1\ ) \\
\vdash & (\ \mathtt{a*+b},\ FT'E',\ 14\ ) \\
\vdash & (\ \mathtt{a*+b},\ \mathtt{a}T'E'\ ,\ 148)
\end{array}
$$

$$
\begin{array}{ll}
\vdash & (\mathtt{*+b},\ T'E'\ \ \ ,\ 148\ \ \ \ \ \ \ \ ) \\
\vdash & (\mathtt{*+b},\ *FT'E',\ 1485\ \ \ \ \ ) \\
\vdash & (\ \mathtt{+b},\ FT'E'\ \ ,\ 1485\ \ \ \ \ ) \\
\vdash & (\ \mathtt{+b},\ T'E'\ \ \ ,\ 1485\ \ \ \ \ ) \\
\vdash & (\ \mathtt{+b},\ E'\ \ \ \ \ ,\ 14856\ \ \ \ ) \\
\vdash & (\ \mathtt{+b},\ +TE'\ \ \ ,\ 148562\ \ \ ) \\
\vdash & (\ \ \mathtt{b},\ TE'\ \ \ \ ,\ 148562\ \ \ ) \\
\vdash & (\ \ \mathtt{b},\ FT'E'\ \ ,\ 1485624\ \ ) \\
\vdash & (\ \ \mathtt{b},\ \mathtt{b}T'E'\ \ ,\ 14856249\ \ ) \\
\vdash & (\ \ \varepsilon,\ T'E'\ \ \ ,\ 14856249\ \ ) \\
\vdash & (\ \ \varepsilon,\ E'\ \ \ \ \ ,\ 148562496\ ) \\
\vdash & (\ \ \varepsilon,\ \varepsilon\ \ \ \ \ \ ,\ 1485624963)
\end{array}
$$