IndentationSensitiveLanguages

Leonhard Brunauer Bernhard Mühlbacher

University of Salzburg

June 29, 2006

Outline

Description

- Spin-off of the project we are working on in the compiler construction class: Boa
- Nested structures are not defined by BEGIN and END tokens, but by the level of indentation of statements.
- This feature can be found in several other programming languages around:
  - Haskell
  - Python

Example

```
def sign(i: int) -> int:
    if i < 0:
        return -1
    elif i == 0:
        return 0
    else:
        return 1
```

Definition

Definition

Let Σ be an alphabet. A *counter* C_t^n for some symbol $t \in \Sigma$ and some number $n \in \mathbb{N}$ is defined by:

1. $C_t^0 = \varepsilon$ and
2. $C_t^{n+1} = C_t^n \circ t$, where $\circ$ is the concatenation symbol.

Referring to the above definition, a counter obviously generates a string of length n that does not contain any other symbol than t.

Definition (cond't)

Definition

An indentation sensitive grammar *ISG* is a 5-tuple (V, Σ, C, R, S), where

1. V is a finite set of *non-terminal* symbols,
2. Σ is a finite set of *terminal* symbols,
3. C is a set of *counters*

$$C = \{C_t^n \mid t \in \Sigma \text{ and } n \in \mathbb{N}\}$$

4. R is a finite set of *rules*, where the left-hand side is a string of zero or one counter and exactly one non-terminal and the right-hand side is a string of non-terminals, terminals, and counters, and
5. $S \in V$ is the *start symbol*.

Example

The grammar for a simplified programming language is defined by $G = (V, \Sigma, C, R, S)$, where

1. $V = \{S, \langle Stmt \rangle, \langle Simple \rangle, \langle Nesting \rangle, \langle Cond \rangle\}$
2. $\Sigma = \{\text{“if “}, \text{identifier}, \text{intLiteral}, \text{“ : “}, \text{“ = “}, \text{newline}, \rightarrow\}$
3. $C = \{C^n_{\rightarrow} \mid n \in \mathbb{N}\}$
4. R is defined by:

$$
\begin{aligned}
S &\rightarrow C^0_{\rightarrow}\langle Stmt \rangle \mid \varepsilon \\
C^n_{\rightarrow}\langle Stmt \rangle &\rightarrow C^n_{\rightarrow}\langle Stmt \rangle C^n_{\rightarrow}\langle Stmt \rangle \mid C^n_{\rightarrow}\langle Nesting \rangle \mid C^n_{\rightarrow}\langle Simple \rangle \\
C^n_{\rightarrow}\langle Nesting \rangle &\rightarrow C^n_{\rightarrow} \text{“if“ } \langle Cond \rangle \text{ “ : “ newline } C^{n+1}_{\rightarrow}\langle Stmt \rangle \\
\langle Simple \rangle &\rightarrow \text{identifier “ = “ intLiteral newline} \\
\langle Cond \rangle &\rightarrow \text{identifier}
\end{aligned}
$$

Pumping Lemma for context-free Languages (M.Sipser)

If A is a context-free language, then there is a number p(pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^i xy^i z \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

Theorem + Proof

Theorem

G is not context free.

Proof.

We assume that G is a CFL and obtain a contradiction. Let p be the pumping length for G that is guaranteed to exist by the pumping lemma. We use the string
$s = if \circ \rightarrow \circ if \circ \rightarrow \circ \rightarrow \circ if \circ \rightarrow \circ \rightarrow \circ \rightarrow \circ if \circ .... \circ (\rightarrow)^p \circ ass$ or an other representation:

$$
\begin{aligned}
s = \; & if \circ \\
& \rightarrow \circ if \circ \\
& \rightarrow \circ \rightarrow \circ if \circ \\
& \rightarrow \circ \rightarrow \circ \rightarrow \circ if \circ \\
& .... \\
& \underbrace{\rightarrow \circ ..... \circ \rightarrow}_{p \text{ times}} \circ ass
\end{aligned}
$$

Proof (cond't)

- Clearly s is a member of G and of length at least p.
- The pumping lemma states that s can be pumped, but we show that it cannot.
- In other words, we show that no matter how we choose v one of the three conditions of the lemma is violated.

Proof (cond't)

1. $vxy =$ "*if*": pumping v leads to a string like $if \circ if$... and such a string is not in G, because after each *if* we have to increase the indentation level (one $\rightarrow$ more than the *if* before).
2. $vxy =$ "$if \circ \rightarrow$": pumping v we also get a string like $if \circ \rightarrow \circ if \circ \rightarrow \circ if \circ ...$, that is not in G, if $i > 1$ (pump more than once). Here we again violates to increase the indentation level after each *if*.
3. $vxy =$ "$\rightarrow$": pumping v again results in a string $if \circ \rightarrow \circ \rightarrow \circ ...$, which is not in G. We have to increase the indentation level after each if **exactly** once.
4. $vxy =$ "*ass*": here the condition 1 of the pumping lemma is violated if $i = 0$, because we need an assignment *ass*
5. $vxy =$ "$\underbrace{\rightarrow \circ ... \circ \rightarrow}_{p \text{ times}} \circ ass$": if $i = 0$, condition 1 is violated (no *ass*) and also condition 3 ($|vxy| > p$)

Limitation of the indentation level

Theorem

The depth of indentation level is finite, and is limited by the length of the input string.

Proof.

This theorem is obviously true because the number of $\rightarrow$'s cannot be greater than the input string. $\square$

For our example we have a lower bound of $O(\sqrt{n})$ where $n = |s|$

$$
\begin{aligned}
s = \; & if \circ \\
& \rightarrow \circ if \circ \\
& \rightarrow \circ \rightarrow \circ if \circ \\
& \underbrace{\rightarrow \circ \rightarrow \circ \rightarrow \circ if \circ}_{m}
\end{aligned}
$$

$n = \sum_{i=1}^{m} i = \frac{m*(m+1)}{2} = \frac{(m^2+m)}{2}$.

Parser construction

Try to construct an efficient LL-parser for an indentation sensitive grammar.

1. Eliminating left recursion
2. Counter binding
3. Getting rid of backtracking

Eliminating left recursion

- Left recursion may lead to nondeterministic choices or prevent the parser from terminating.

$$A \rightarrow A\alpha \mid \beta$$

- Left recursion can be eliminated by transforming the grammar to

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}$$

Counter binding

- Simplify the choice among deriving a variable only and deriving a variable and a counter at once.
- Prevent from nondeterminism.

Example

Let's try to find a derivation for $C_t^n A$

$$\begin{aligned} C_t^n A &\rightarrow C_t^n B \\ C_t^n B &\rightarrow C_t^n B_1 \\ B &\rightarrow B_2 \end{aligned}$$

Which rule to apply for $C_t^n B$?

Counter binding (cont'd)

- To get rid of this choice we introduce counter binding.
  - We can explicitly bind counters and variables to make them behave like a single variable.
  - A counter that is not bounded is treated like a terminal symbol.
  - If a counter appears on a LHS, it must be bounded.

Example

$$\begin{aligned} \overline{C_t^n A} &\rightarrow \overline{C_t^n B} \\ \overline{C_t^n B} &\rightarrow C_t^n B_1 \\ B &\rightarrow B_2 \end{aligned}$$

$$\overline{C_t^n A} \Rightarrow \overline{C_t^n B} \Rightarrow C_t^n B_1$$

Getting rid of backtracking

- Introducing counters may lead to nondeterministic choices.

$$\overline{C_t^n A} \quad \rightarrow \quad \overline{C_t^n B_1} \mid \overline{C_t^n B_2}$$

- $FIRST(\overline{C_t^n B_1}) = FIRST(\overline{C_t^n B_2})$ if $n \neq 0$
- Delay the choice and fetch n t-symbols first.
- Deterministic choice, iff $FIRST(B_1) \neq FIRST(B_2)$

Getting rid of backtracking (cont'd)

- The situation gets more complicated if we have

$$\overline{C_t^n A} \quad \rightarrow \quad \overline{C_t^{n+c} B_1} \mid \overline{C_t^n B_2}$$

  for some $c \in \mathbb{N}$
- Introduce a new counter field *value*.
- Say that a counter C_t^n is *applied* to the input, if we try to fetch t-tokens until $value = n$.
- Apply C_t^{n+c} to the input:
  - If the counter's $value = n$, proceed with $\overline{C_t^{n+c} B_1}$
  - Otherwise set C_t^n's value to C_t^{n+c}'s one and try to proceed with $\overline{C_t^n B_2}$

Demo

Efficient Parser Implementation vs Theory

- Counters are placed on the parser's stack.
- Counters can grow to an arbitrary large number, dependent on the length of the input string.
- From a theoretical point of view, every stack element needs an unbounded amount of memory.
- From a pragmatic point of view, we can assume that a stack element needs only a bounded amount of memory for every reasonable input string.

Thanks for your attention!