Script generated by TTT

Title: Petter: Compiler Construction (21.05.2020)
-17: RLL(1) Parsers

Date: Wed May 06 10:42:52 CEST 2020

Duration: 46:07 min

Pages: 18

Right-Regular Context-Free Parsing

Recurring scheme in programming languages: Lists of sth...

$S \to b \quad | \quad S\,a\,b$

Alternative idea: Regular Expressions

$S \to \boxed{(\,b\,a\,)^*\,b}$

Right-Regular Context-Free Parsing

Recurring scheme in programming languages: Lists of sth...

$S \to b \quad | \quad S\,a\,b$

Alternative idea: Regular Expressions

$S \to (\,b\,a\,)^*\,b$

Definition: Right-Regular Context-Free Grammar

A right-regular context-free grammar (RR-CFG) is a 4-tuple $G = (N, T, P, S)$ with:

- $\boxed{N}$ the set of nonterminals,
- $\boxed{T}$ the set of terminals,
- P the set of rules with regular expressions of symbols as rhs,
- $\boxed{S \in N}$ the start symbol

Right-Regular Context-Free Parsing

Recurring scheme in programming languages: Lists of sth...

$S \to b \quad | \quad S\,a\,b$

Alternative idea: Regular Expressions

$S \to (\,b\,a\,)^*\,b$

Definition: Right-Regular Context-Free Grammar

A right-regular context-free grammar (RR-CFG) is a 4-tuple $G = (N, T, P, S)$ with:

- N the set of nonterminals,
- T the set of terminals,
- P the set of rules with regular expressions of symbols as rhs,
- $S \in N$ the start symbol

Example: Arithmetic Expressions

$$\begin{array}{lcl} S & \to & E \\ E & \to & T\,\boxed{(\,+\,T\,)^*} \\ T & \to & F\,\boxed{(\,*\,F\,)^*} \\ F & \to & (\,E\,) \mid \text{name} \mid \text{int} \end{array}$$

Idea 1: Rewrite the rules from G to $\langle G\rangle$:

$$\begin{array}{lcll}
A & \to & \langle\alpha\rangle & \text{if } A \to \alpha \in P \\
\langle\alpha\rangle & \to & \alpha & \text{if } \alpha \in N \cup T \\
\langle\epsilon\rangle & \to & \epsilon & \\
\langle\alpha^*\rangle & \to & \epsilon \mid \langle\alpha\rangle\langle\alpha^*\rangle & \text{if } \alpha \in \text{Regex}_{T,N} \\
\langle\alpha_1 \dots \alpha_n\rangle & \to & \langle\alpha_1\rangle \dots \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N} \\
\langle\alpha_1 \mid \dots \mid \alpha_n\rangle & \to & \langle\alpha_1\rangle \mid \dots \mid \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N}
\end{array}$$

…and generate the according LL(k)-Parser $M^L_{\langle G\rangle}$

Idea 1: Rewrite the rules from G to $\langle G\rangle$:

$$\begin{array}{lcll}
A & \to & \langle\alpha\rangle & \text{if } A \to \alpha \in P \\
\langle\alpha\rangle & \to & \alpha & \text{if } \alpha \in N \cup T \\
\langle\epsilon\rangle & \to & \epsilon & \\
\langle\alpha^*\rangle & \to & \epsilon \mid \langle\alpha\rangle\langle\alpha^*\rangle & \text{if } \alpha \in \text{Regex}_{T,N} \\
\langle\alpha_1 \dots \alpha_n\rangle & \to & \langle\alpha_1\rangle \dots \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N} \\
\langle\alpha_1 \mid \dots \mid \alpha_n\rangle & \to & \langle\alpha_1\rangle \mid \dots \mid \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N}
\end{array}$$

…and generate the according LL(k)-Parser $M^L_{\langle G\rangle}$

Example: Arithmetic Expressions cont'd

$$\begin{array}{lcl}
S & \to & E \\
E & \to & T\,(\,+T\,)^* \\
T & \to & F\,(\,*F\,)^* \\
F & \to & (\,E\,) \mid \text{name} \mid \text{int}
\end{array}$$

Idea 1: Rewrite the rules from G to $\langle G\rangle$:

$$\begin{array}{lcll}
A & \to & \langle\alpha\rangle & \text{if } A \to \alpha \in P \\
\langle\alpha\rangle & \to & \alpha & \text{if } \alpha \in N \cup T \\
\langle\epsilon\rangle & \to & \epsilon & \\
\langle\alpha^*\rangle & \to & \epsilon \mid \langle\alpha\rangle\langle\alpha^*\rangle & \text{if } \alpha \in \text{Regex}_{T,N} \\
\langle\alpha_1 \dots \alpha_n\rangle & \to & \langle\alpha_1\rangle \dots \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N} \\
\langle\alpha_1 \mid \dots \mid \alpha_n\rangle & \to & \langle\alpha_1\rangle \mid \dots \mid \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N}
\end{array}$$

…and generate the according LL(k)-Parser $M^L_{\langle G\rangle}$

Example: Arithmetic Expressions cont'd

$$\begin{array}{lcl}
S & \to & E \\
E & \to & \langle T\,(\,+T\,)^*\rangle \\
T & \to & F\,(\,*F\,)^* \\
F & \to & (\,E\,) \mid \text{name} \mid \text{int} \\
\langle T\,(\,+T\,)^*\rangle & \to & T\,\langle(\,+T\,)^*\rangle
\end{array}$$

Idea 1: Rewrite the rules from G to $\langle G\rangle$:

$$\begin{array}{lcll}
A & \to & \langle\alpha\rangle & \text{if } A \to \alpha \in P \\
\langle\alpha\rangle & \to & \alpha & \text{if } \alpha \in N \cup T \\
\langle\epsilon\rangle & \to & \epsilon & \\
\langle\alpha^*\rangle & \to & \epsilon \mid \langle\alpha\rangle\langle\alpha^*\rangle & \text{if } \alpha \in \text{Regex}_{T,N} \\
\langle\alpha_1 \dots \alpha_n\rangle & \to & \langle\alpha_1\rangle \dots \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N} \\
\langle\alpha_1 \mid \dots \mid \alpha_n\rangle & \to & \langle\alpha_1\rangle \mid \dots \mid \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N}
\end{array}$$

…and generate the according LL(k)-Parser $M^L_{\langle G\rangle}$

Example: Arithmetic Expressions cont'd

$$\begin{array}{lcl}
S & \to & E \\
E & \to & \langle T\,(\,+T\,)^*\rangle \\
T & \to & F\,(\,*F\,)^* \\
F & \to & (\,E\,) \mid \text{name} \mid \text{int} \\
\langle T\,(\,+T\,)^*\rangle & \to & T\,\langle(\,+T\,)^*\rangle \\
\langle(\,+T\,)^*\rangle & \to & \epsilon \mid \langle +T\rangle\langle(\,+T\,)^*\rangle
\end{array}$$

Idea 1: Rewrite the rules from G to $\langle G \rangle$:

$$\begin{array}{llll}
A & \to & \langle\alpha\rangle & \text{if } A \to \alpha \in P \\
\langle\alpha\rangle & \to & \alpha & \text{if } \alpha \in N \cup T \\
\langle\epsilon\rangle & \to & \epsilon & \\
\langle\alpha^*\rangle & \to & \epsilon \mid \langle\alpha\rangle\langle\alpha^*\rangle & \text{if } \alpha \in \text{Regex}_{T,N} \\
\langle\alpha_1 \dots \alpha_n\rangle & \to & \langle\alpha_1\rangle \dots \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N} \\
\langle\alpha_1 \mid \dots \mid \alpha_n\rangle & \to & \langle\alpha_1\rangle \mid \dots \mid \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N}
\end{array}$$

... and generate the according LL(k)-Parser $M^L_{\langle G \rangle}$

Example: Arithmetic Expressions cont'd

$$\begin{array}{lll}
S & \to & E \\
E & \to & \langle T\,(+T)^*\rangle \\
T & \to & \boxed{F\,(*F)^*} \\
F & \to & (\,E\,) \mid \text{name} \mid \text{int} \\
\langle T\,(+T)^*\rangle & \to & T\,\langle(+T)^*\rangle \\
\langle(+T)^*\rangle & \to & \epsilon \mid \langle +T\rangle\langle(+T)^*\rangle \\
\langle +T\rangle & \to & +T
\end{array}$$

Idea 1: Rewrite the rules from G to $\langle G \rangle$:

$$\begin{array}{llll}
A & \to & \langle\alpha\rangle & \text{if } A \to \alpha \in P \\
\langle\alpha\rangle & \to & \alpha & \text{if } \alpha \in N \cup T \\
\langle\epsilon\rangle & \to & \epsilon & \\
\langle\alpha^*\rangle & \to & \epsilon \mid \langle\alpha\rangle\langle\alpha^*\rangle & \text{if } \alpha \in \text{Regex}_{T,N} \\
\langle\alpha_1 \dots \alpha_n\rangle & \to & \langle\alpha_1\rangle \dots \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N} \\
\langle\alpha_1 \mid \dots \mid \alpha_n\rangle & \to & \langle\alpha_1\rangle \mid \dots \mid \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N}
\end{array}$$

... and generate the according LL(k)-Parser $M^L_{\langle G \rangle}$

Example: Arithmetic Expressions cont'd

$$\begin{array}{lll}
S & \to & E \\
E & \to & \langle T\,(+T)^*\rangle \\
T & \to & \boxed{\langle F\,(*F)^*\rangle} \\
F & \to & (\,E\,) \mid \text{name} \mid \text{int} \\
\langle T\,(+T)^*\rangle & \to & T\,\langle(+T)^*\rangle \\
\langle(+T)^*\rangle & \to & \epsilon \mid \langle +T\rangle\langle(+T)^*\rangle \\
\langle +T\rangle & \to & +T
\end{array}$$

Idea 1: Rewrite the rules from G to $\langle G \rangle$:

$$\begin{array}{llll}
A & \to & \langle\alpha\rangle & \text{if } A \to \alpha \in P \\
\langle\alpha\rangle & \to & \alpha & \text{if } \alpha \in N \cup T \\
\langle\epsilon\rangle & \to & \epsilon & \\
\langle\alpha^*\rangle & \to & \epsilon \mid \langle\alpha\rangle\langle\alpha^*\rangle & \text{if } \alpha \in \text{Regex}_{T,N} \\
\langle\alpha_1 \dots \alpha_n\rangle & \to & \langle\alpha_1\rangle \dots \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N} \\
\langle\alpha_1 \mid \dots \mid \alpha_n\rangle & \to & \langle\alpha_1\rangle \mid \dots \mid \langle\alpha_n\rangle & \text{if } \alpha_i \in \text{Regex}_{T,N}
\end{array}$$

... and generate the according LL(k)-Parser $M^L_{\langle G \rangle}$

Example: Arithmetic Expressions cont'd

$$\begin{array}{lll}
S & \to & E \\
E & \to & \boxed{\langle T\,(+T)^*\rangle} \equiv A \\
T & \to & \cancel{\langle F\,(*F)^*\rangle} \equiv B \\
F & \to & (\,E\,) \mid \text{name} \mid \text{int} \\
A\ \cancel{\langle T\,(+T)^*\rangle} & \to & T\,\cancel{\langle(+T)^*\rangle} \equiv C \\
C\ \cancel{\langle(+T)^*\rangle} & \to & \epsilon \mid D\cancel{\langle +T\rangle}\cancel{\langle(+T)^*\rangle}\ C \\
D\ \cancel{\langle +T\rangle} & \to & +T \\
B\ \cancel{\langle F\,(*F)^*\rangle} & \to & F\,\langle(*F)^*\rangle \\
\langle(*F)^*\rangle & \to & \epsilon \mid \langle *F\rangle\,\langle(*F)^*\rangle \\
\langle *F\rangle & \to & *F
\end{array}$$

Reinhold Heckmann

Definition:

An $RR\text{-}CFG\ G$ is called $\boxed{RLL(1)}$,
if the corresponding CFG $\boxed{\langle G \rangle}$ is an $\boxed{LL(1) \text{ grammar.}}$

Discussion

- directly yields the table driven parser $M^L_{\langle G \rangle}$ for $RLL(1)$ grammars
- however: mapping regular expressions to recursive productions unnessessarily strains the stack
  $\to$ instead directly construct automaton in the style of Berry-Sethi

Idea 2: Recursive Descent RLL Parsers:

Recursive descent RLL(1)-parsers are an alternative to table-driven parsers; apart from the usual function `scan()`, we generate a program frame with the lookahead function `expect()` and the main parsing method `parse()`:

void

```
int next;
boolean expect(Set E){
    if ({ε,next} ∩ E = ∅){
        cerr << "Expected" << E << "found" << next;
        exit(0);
    }
    return ;
}
void parse(){
    next = scan();
    expect(First1(S));
    S();
    expect({EOF}) ;
}
```

52 / 55

Idea 2: Recursive Descent RLL Parsers:

For each $A \to \alpha \in P$, we introduce:

```
void A(){
    generate(α)
}
```

with the meta-program $generate$ being defined by structural decomposition of α:

$$
\begin{array}{lll}
generate(r_1 \ldots r_k) & = & generate(r_1) \\
 & & \texttt{expect}(\mathsf{First}_1(r_2)) \; ; \\
 & & generate(r_2) \\
 & & \vdots \\
 & & \texttt{expect}(\mathsf{First}_1(r_k)) \; ; \\
 & & generate(r_k) \\
generate(\epsilon) & = & ; \\
generate(a) & = & \texttt{next} = \texttt{scan}(); \\
generate(A) & = & \texttt{A}(); \\
\end{array}
$$

53 / 55

Idea 2: Recursive Descent RLL Parsers:

$$
\begin{array}{lll}
generate(r^*) & = & \texttt{while} \; (\; \texttt{next} \in \mathsf{F}_\epsilon(r)) \; \{ \\
 & & \quad generate(r) \\
 & & \} \\
generate(r_1 \mid \ldots \mid r_k) & = & \texttt{switch}(\texttt{next}) \; \{ \\
 & & \quad labels(\mathsf{First}_1(r_1)) \; generate(r_1) \; \texttt{break} \; ; \\
 & & \quad \vdots \\
 & & \quad labels(\mathsf{First}_1(r_k)) \; generate(r_k) \; \texttt{break} \; ; \\
 & & \} \\
labels(\{\alpha_1, \ldots, \alpha_m\}) & = & label(\alpha_1): \; \ldots \; label(\alpha_m): \\
label(\alpha) & = & \texttt{case} \; \alpha \\
label(\epsilon) & = & \texttt{default}
\end{array}
$$

54 / 55

Topdown-Parsing

Discussion

- A practical implementation of an $RLL(1)$-parser via recursive descent is a straight-forward idea
- However, only a subset of the deterministic contextfree languages can be parsed this way.
- As soon as $\mathsf{First}_1(\_)$ sets are not disjoint any more,

55 / 55

Idea 2: Recursive Descent RLL Parsers:

$$
\begin{array}{lcl}
generate(r^*) & = & \texttt{while}\ (\ \texttt{next} \in \mathsf{F}_\epsilon(r))\ \{ \\
 & & \quad generate(r) \\
 & & \} \\
generate(r_1 \mid \ldots \mid r_k) & = & \texttt{switch}(\texttt{next})\ \{ \\
 & & \quad labels(\boxed{\mathsf{First}_1(r_1)})\ generate(r_1)\ \texttt{break}\ ; \\
 & & \quad \vdots \\
 & & \quad labels(\boxed{\mathsf{First}_1(r_k)})\ generate(r_k)\ \texttt{break}\ ; \\
 & & \} \\
labels(\{\alpha_1, \ldots, \alpha_m\}) & = & label(\alpha_1)\texttt{:}\ \ldots\ label(\alpha_m)\texttt{:} \\
label(\alpha) & = & \texttt{case}\ \alpha \\
label(\epsilon) & = & \texttt{default}
\end{array}
$$

Topdown-Parsing

Discussion

- A practical implementation of an $RLL(1)$-parser via recursive descent is a straight-forward idea
- However, only a subset of the deterministic contextfree languages can be parsed this way.
- As soon as $\mathsf{First}_1(\_)$ sets are not disjoint any more,

Topdown-Parsing

Discussion

- A practical implementation of an $RLL(1)$-parser via recursive descent is a straight-forward idea
- However, only a subset of the deterministic contextfree languages can be parsed this way.
- As soon as $\mathsf{First}_1(\_)$ sets are not disjoint any more,
  - Solution 1: For many accessibly written grammars, the alternation between right hand sides happens too early. Keeping the common prefixes of right hand sides joined and introducing a new production for the actual diverging sentence forms often helps.
  - Solution 2: Introduce *ranked* grammars, and decide conflicting lookahead always in favour of the higher ranked alternative
    $\to$ relation to LL parsing not so clear any more
    $\to$ not so clear for $\_^*$ operator how to decide
  - Solution 3: Going from $LL(1)$ to $LL(k)$
    The size of the occuring sets is rapidly increasing with larger k
    Unfortunately, even $LL(k)$ parsers are not sufficient to accept all deterministic contextfree languages. (regular lookahead $\to$ $LL(*)$)
- In practical systems, this often motivates the implementation of $k = 1$ only ...