

LL(1) Parsing, Refactoring and Recursive Descent

CSCI 3136: Principles of
Programming Languages

Agenda

- Building an LL(1) Parser
- The PREDICT Table
- Constructing FIRST, FOLLOW, and PREDICT
- Is a Grammar LL(1)?
- Refactoring
- Recursive Descent

Building an LL(1) Parser

- Basic Challenge: Given current token, which production does the parser select if next item in sentential form is a nonterminal

E.g., if S is on the stack and input is $+$, then parser must select production $S \rightarrow +SS$

- In general: for input \mathbf{a} and sentential form A , either
 - $A \sqsupseteq \alpha \sqsupseteq X\mathbf{a}\beta$
 - $A \sqsupseteq \alpha \sqsupseteq X\epsilon$ and derivation of A is succeeded by \mathbf{a} .
- Intuitively, \mathbf{a} is in the *predictor set* of $A \rightarrow \alpha$
if $A\beta \sqsupseteq \alpha\beta \sqsupseteq Xa\gamma$, for $\beta, \gamma \in \Sigma^*$

LL(1) Grammars

- **Definition:** A grammar is LL(1) if the predictor sets of all productions with the same LHS are disjoint.
- E.g. S-Grammars are LL(1)

Grammar

1. $S \rightarrow + SS$
2. $S \rightarrow - SS$

PREDICT Table

Production	Predictor Set
$S \rightarrow + S S$	{+}
$S \rightarrow - S S$	{-}
$S \rightarrow * S S$	{*}
$S \rightarrow / S S$	{/}
$S \rightarrow \text{neg } S$	{neg}
$S \rightarrow \text{integer}$	{integer}

Constructing PREDICT: The 3 Tables

- **FIRST(\square):** the set of leftmost terminals **a** that can be derived from $\square \in (V \cup \Sigma)^*$

$$\alpha \in X a \beta$$

- **FOLLOW(X):** the set of the first terminals **a** that immediately follow variable X in a derivation

$$S \in X \alpha X a \beta$$

- **PREDICT($A \rightarrow \alpha$):** the set of terminals that predict this production given **A**

The FIRST Table

- **Definition:** $\text{FIRST}(\sigma)$, $\forall \sigma \in (V \cup \Sigma)$:
 - For $a \in \Sigma$, $a \in \text{FIRST}(\sigma)$ if $\sigma \sqsubseteq X a \beta$
 - $\varepsilon \in \text{FIRST}(\sigma)$ if $\sigma \sqsubseteq X \varepsilon$
- **Idea:** For a sentential form σ , $\text{FIRST}(\sigma)$ is the set of all terminals that could start any future sentential form derived from σ
- **Notes:**
 - For $a \in \Sigma$, $\text{FIRST}(a) = \{a\}$
 - Precompute $\text{FIRST}(X)$ only for $X \in V$
 - Generate $\text{FIRST}(\sigma)$, $\sigma \in (V \cup \Sigma)^*$ as needed

The FIRST Table (Part 2)

To Compute **FIRST** (for a grammar)

- For $a \in \Sigma$, $\text{FIRST}(a) = \{a\}$
- For $X \in V$, $\text{FIRST}(X) = A$
- Repeat until no new additions to $\text{FIRST}(X)$, $X \in V$ are possible:

if $X \rightarrow \alpha \in P$,

$$\text{FIRST}(X) = \text{FIRST}(X) \cup \text{FIRST}(\alpha)$$

Note, **FIRST(α)** (First for sentential forms)

- $\alpha = \alpha_1\alpha_2\ldots\alpha_k$, $\alpha_i \in (V \cup \Sigma)$
- $\text{FIRST}(\alpha) = A$

The FIRST Table: Example

Grammar	Symbol	Iter. 0	Iter. 1	Iter.2	FIRST
<ul style="list-style-type: none"> $T \rightarrow AB$ $A \rightarrow PQ_{\Sigma}$ $A \rightarrow BC$ $P \rightarrow pP$ $P \rightarrow \varepsilon$ $Q \rightarrow qQ_V$ $Q \rightarrow \varepsilon$ $B \rightarrow bB$ 	p	{p}	{p}	{p}	{p}
	q	{q}	{q}	{q}	{q}
	b	{b}	{b}	{b}	{b}
	e	{e}	{e}	{e}	{e}
	c	{c}	{c}	{c}	{c}
	f	{f}	{f}	{f}	{f}
	T	A	A	A	{p,q,b,e}
	A	A	A	{p,q,b,e, \square }	{p,q,b,e, \square }
	P	A	{p, \square }	{p, \square }	{p, \square }
	Q	A	{q, \square }	{q, \square }	{q, \square }
	B	A	{b,e}	{b,e}	{b,e}
	C	A	{c,f}	{c,f}	{c,f}

The FOLLOW Table

- **Definition:** $\text{FOLLOW}(X)$, $\forall X \in V$:
 - For $a \in \Sigma$, $a \in \text{FOLLOW}(X)$ if $S \sqsupseteq X \alpha X a \beta$
 - $\epsilon \in \text{FOLLOW}(X)$ if $S \sqsupseteq X \alpha X$
- **Idea:** The FOLLOW set of a variable is the set of all terminals that can occur after that variable (i.e., immediately to the right) in any sentential form
- **To Compute FOLLOW**
 - $\text{FOLLOW}(S) = \{\epsilon\}$
 - For $X \in V$, $\text{FOLLOW}(X) = A$

Repeat until no new additions to $\text{FOLLOW}(X)$, $X \in V$

The FOLLOW Table: Example

Grammar

- $T \rightarrow AB$
- $A \rightarrow PQ$
- $A \rightarrow BC$
- $P \rightarrow pP$
- $P \rightarrow \epsilon$
- $Q \rightarrow qQ$
- $Q \rightarrow \epsilon$
- $B \rightarrow bB$

Symbol	Iter. 0	Iter. 1	FOLLOW
ϵ	$\{\epsilon\}$	$\{\epsilon\}$	
T	A	{b,e}	$\{\epsilon\}$
A	A	{q}	{b,e}
P	A	A	{q,b,e}
Q	A	$\{\epsilon, c, f\}$	{b,e}
B	A	A	$\{\epsilon, c, f\}$
C			{b,e}

1 $X \rightarrow \alpha A \beta \in P$,

$FOLLOW(A) = FOLLOW(A) \cup (FIRST(\beta) - \{\epsilon\})$

if $\epsilon \in FIRST(\beta)$ then $FOLLOW(A) = FOLLOW(A) \cup FOLLOW(X)$

To find the FOLLOW set for A, find productions with A on the right hand side:

- For each production $X \rightarrow \alpha A \beta$, put $FIRST(\beta) - \{\epsilon\}$ in $FOLLOW(A)$
- If ϵ is in $FIRST(\beta)$ then put $FOLLOW(X)$ into $FOLLOW(A)$
- For each production $X \rightarrow \alpha A$, put $FOLLOW(X)$ into $FOLLOW(A)$

FIRST	
p	{p}
q	{q}
b	{b}
e	{e}
c	{c}
f	{f}
T	{p,q,b,e}
A	{p,q,b,e, ϵ }
P	{p, ϵ }
Q	{q, ϵ }
B	{b,e}
C	{c,f}

The PREDICT Table

- **Definition:** For $a \in \Sigma \cup \{\epsilon\}$, $a \in \text{PREDICT}(A \rightarrow \alpha)$ if
 - $a \in \text{FIRST}(\alpha) - \{\epsilon\}$ or
 - $\epsilon \in \text{FIRST}(\alpha)$ and $a \in \text{FOLLOW}(A)$
- **Idea:** The predict set of terminal symbols for a production is the FIRST set of the RHS plus the FOLLOW set of the production if ϵ is part of the FIRST set
- **To Compute PREDICT**
 - For each $(A \rightarrow \alpha) \in P$, $\text{PREDICT}(A \rightarrow \alpha) = A$
 - For each $(A \rightarrow \alpha) \in P$

The PREDICT Table: Example

Symbol	FIRST	FOLLOW
	{p,q,b,e}	{ \square }
T	{p,q,b,e, \square }	{b,e}
A	{p, \square }	{q,b,e}
P	{q, \square }	{b,e}
Q	{b,e}	{ \square ,c,f}
B	{c,f}	{b,e}
C		

For each $(A \rightarrow \alpha) \in P$

$PREDICT(A \rightarrow \alpha) = FIRST(\alpha) - \{\epsilon\}$

if $\epsilon \in FIRST(\alpha)$ then

$PREDICT(A \rightarrow \alpha) = PREDICT(A \rightarrow \alpha) \cup FOLLOW(A)$

Since the predictor sets overlap for A productions, this is not an LL(1) grammar

Production	Predictor Set
$T \rightarrow AB$	{p,q,b,e}
$A \rightarrow PQ$	{p,q,b,e}
$A \rightarrow BC$	{b,e}
$P \rightarrow pP$	{p}
$P \rightarrow \square$	{q,b,e}
$Q \rightarrow qQ$	{q}
$Q \rightarrow \square$	{b,e}
$B \rightarrow bB$	{b}
$B \rightarrow e$	{e}
$C \rightarrow cC$	{c}
$C \rightarrow f$	{f}

How to Prove a Grammar is LL(1)

- Construct PREDICT Table
- This grammar is not LL(1) if and only if there are two productions with the same left hand side have non disjoint predictor sets.
- Note: It's actually possible to build the FIRST, FOLLOW, and PREDICT tables by simply looking at the grammar.
- What happens if our grammar is not LL(1)?

Limitations and Problems with LL(1)

- There exist context free languages that do not have LL(1) grammars
- There is no known algorithm to determine whether a language is LL(1)
- There is an algorithm to decide whether a grammar is LL(1) (we just saw it)
- Most obvious grammars for most programming languages are usually not LL(1)
- In many cases a non-LL(1) grammar can be refactored into an LL(1) grammar

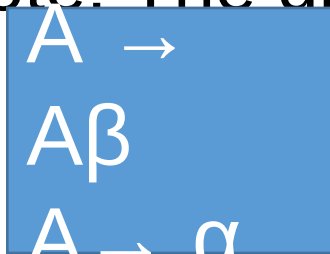
Refactoring Grammars

- Two common problems:
Which production do you use? (Both have α in FIRST)
- Left recursion
 - $A \rightarrow A\beta$
 - $A \rightarrow \alpha$
- Common Prefix
 - $A \rightarrow \alpha\beta$
 - $A \rightarrow \alpha\gamma$

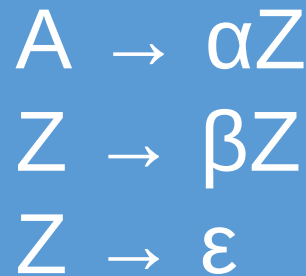
Dealing with Left Recursion

- Idea: Replace Left Recursion with Right Recursion

- Note: The grammar generates $\alpha\beta^*$



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



$A \rightarrow \alpha Z$
 $Z \rightarrow \beta Z$
 $Z \rightarrow \epsilon$

- Note: As a side-effect the grammar may cease to capture some properties such as left-

Example of Eliminating Left Recursion

- Consider the grammar fragment:

Block \rightarrow '{' Statements '}'

Statements \rightarrow Statements Statement

Statements $\rightarrow \epsilon$

- Replace this with:

Block \rightarrow '{' Statements '}'

Statements \rightarrow Statement Statements

Statements $\rightarrow \epsilon$

Dealing with Common Prefix

- Idea: Remove common prefix by left factoring
- Note: This grammar generates $\alpha(\beta|\gamma)$

$$\begin{array}{l} A \rightarrow \alpha\beta \\ A \rightarrow \alpha\gamma \end{array}$$

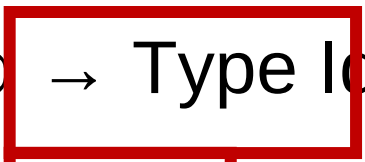


$$\begin{array}{l} A \rightarrow \alpha Y \\ Y \rightarrow \beta \\ Y \rightarrow \gamma \end{array}$$

Example of Eliminating Common Prefix

- Bad Grammar

Field \rightarrow Type Identifier
‘.’



Field \rightarrow Type identifier
‘(‘ Args ‘)’ ‘.’

Type \rightarrow Identifier

Type \rightarrow Identifier Array

Array \rightarrow ‘[‘ ’]’ Array

Array $\rightarrow \epsilon$

- Better grammar

Field \rightarrow Type Identifier
FieldBody ‘.’

FieldBody \rightarrow ‘(‘ Args
‘)’

FieldBody $\rightarrow \epsilon$

Type \rightarrow Identifier Array

Array \rightarrow ‘[‘ ’]’ Array

Array $\rightarrow \epsilon$

LL(1) Parser Implementation

- Two efficient approaches:
 - Recursive Descent
 - Deterministic Pushdown Automata (DPDA)
- Recursive Descent is easier to understand and implement.

Recursive Descent

Idea: For each
variable X, write a
procedure: parse_X()

```
parse_X:
    t = peek_next_token( )

    select X          based on
    t
    for each i       :
        if i == Y1    V:
            parse_Y1( )
        elseif i == Y2
V:
            parse_Y2( )
```

...

Example

parse_S:

t = peek_at_token()

Grammar

• $S \rightarrow \text{Add} \mid \text{Sub} \mid \text{Mul}$

• $S \rightarrow \text{Div} \mid \text{Neg} \mid \text{Val}$

• $\text{Add} \rightarrow + S S$

• $\text{Sub} \rightarrow - S S$

• $\text{Mul} \rightarrow X S S$

• $\text{Div} \rightarrow / S S$

• $\text{Neg} \rightarrow \text{neg } S$

select S based

on t

for each i :

if i == **Add** V:

parse_Add()

elseif i == **Sub**

V:

parse_Sub()

...

elseif i == **Val**