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 → +SS

- In general: for input **a** and sentential form A , either

  - A □ α □ X **a**β
  - A □ α □ X ε and derivation of A is succeeded by **a**.

- Intuitively, **a** is in the *predictor set* of A → α

  if Aβ □ αβ □ X aγ, for β, γ Ε ΣX

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
$S \rightarrow + S\ S$	{+}
$S \rightarrow - S\ S$	{-}
$S \rightarrow * S\ S$	{*}
$S \rightarrow / S\ S$	{/}
$S \rightarrow$ neg S	{neg}
$S \rightarrow$ integer	{integer}

Constructing PREDICT: The 3 Tables

- **FIRST(□):** the set of leftmost terminals **a** that can be derived from □ E (V ν Σ)X

  α □ X **a**β

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

  S □ X αX**a**β

- **PREDICT(A → α):** the set of terminals that predict this production given **A**

The FIRST Table

- **Definition:** FIRST(σ), Ί σ E (V ν Σ):
  - For a E Σ, a E FIRST(σ) if σ □ X aβ
  - ε E FIRST(σ) if σ □ X ε
- **Idea:** For a sentential form σ, FIRST(σ) is the set of all terminals that could start any future sentential form derived from σ
- **Notes:**
  - For a E Σ, FIRST(a) = {a}
  - Precompute FIRST(X) only for X E V
  - Generate FIRST(σ), σ E (V ν Σ)X as needed

The FIRST Table (Part 2)

To Compute FIRST (for a grammar)

- For a E Σ, FIRST(a) = {a}
- For X E V, FIRST(X) = A
- Repeat until no new additions to FIRST(X), X E V are possible:

  Ί X → α E P,

  FIRST(X) = FIRST(X) ν FIRST(α)

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

- α=α1α2...αk, αi E (V ν Σ)
- FIRST(α) = A

The FIRST Table: Example

Grammar

- T → AB
- A → PQΣ
- A → BC
- P → pP
- P → ε
- Q → qQV
- Q → ε
- B → bB

Symbol	Iter. 0	Iter. 1	Iter.2	FIRST
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,□}	{p,q,b,e,□}
P	A	{p,□}	{p,□}	{p,□}
Q	A	{q,□}	{q,□}	{q,□}
B	A	{b,e}	{b,e}	{b,e}
C	A	{c,f}	{c,f}	{c,f}

The FOLLOW Table

- **Definition:** FOLLOW(X),Ί X E V:
  - For a E Σ, a E FOLLOW(X) if S □ X αXaβ
  - ε E FOLLOW(X) if S □ X αX
- **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**
  - FOLLOW(S) = {ε}
  - For XEV, FOLLOW(X) = A
  - Repeat until no new additions to FOLLOW(X), X E

The FOLLOW Table: Example

Grammar

- T → AB
- A → PQ
- A → BC
- P → pP
- P → ε
- Q → qQ
- Q → ε
- B → bB

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

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

Ί X → αAβ E P,

FOLLOW(A) = FOLLOW(A) ν (FIRST(β)-{ε})

if ε E FIRST(β) then FOLLOW(A) = FOLLOW(A) ν FOLLOW(X)

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

- For each production X → αAβ, put FIRST(β) − {ε} in FOLLOW(A)
- If ε is in FIRST(β) then put FOLLOW(X) into FOLLOW(A)
- For each production X → αA, put FOLLOW(X) into FOLLOW(A)

The PREDICT Table

- **Definition:** For a E Σ ν {ε}, a E PREDICT(A → α) if
  - a E FIRST(α)-{ε} or
  - ε E FIRST(α) and a E 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 → α) E P, PREDICT(A → α) = A
  - For each (A → α) E P
    - PREDICT(A → α) = FIRST(α)-{ε}

The PREDICT Table: Example

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

For each (A → α) E P
PREDICT(A → α) = FIRST(α)-{ε}
if ε E FIRST(α) then
PREDICT(A → α) = PREDICT(A → α) ν FOLLOW(A)

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

Production	Predictor Set
T → AB	{p,q,b,e}
A → PQ	{p,q,b,e}
A → BC	{b,e}
P → pP	{p}
P → □	{q,b,e}
Q → qQ	{q}
Q → □	{b,e}
B → bB	{b}
B → e	{e}
C → cC	{c}
C → 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 αβ

$A \to A\beta$

$A \to \alpha$

$A \to \alpha Z$

$Z \to \beta Z$

$Z \to \varepsilon$

- 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 → ‘{‘ Statements ‘}’

  Statements → Statements Statement

  Statements → ε

- Replace this with:

  Block → ‘{‘ Statements ‘}’

  Statements → Statement Statements

  Statements → ε

Dealing with Common Prefix

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

$$\begin{aligned} A &\rightarrow \alpha\beta \\ A &\rightarrow \alpha\gamma \end{aligned} \quad \Longrightarrow \quad \begin{aligned} A &\rightarrow \alpha Y \\ Y &\rightarrow \beta \\ Y &\rightarrow \gamma \end{aligned}$$

Example of Eliminating Common Prefix

- Bad Grammar

Field → Type Identifier ‘;’

Field → Type identifier ‘(‘ Args ‘)’ ‘;’

Type → Identifier

Type → Identifier Array

Array → ’[‘ ’]’ Array

Array → ε

- Better grammar

Field → Type Identifier FieldBody ‘;’

FieldBody → ‘(‘ Args ‘)’

FieldBody → ε

Type → Identifier Array

Array → ’[‘ ’]’ Array

Array → ε

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

Grammar

- S → Add | Sub | Mul
- S → Div | Neg | Val
- Add → + S S
- Sub → − S S
- Mul → X S S
- Div → / S S
- Neg → neg S

```
parse_S:
  t = peek_at_token()

  select S                based
on t
  for each    i             :
    if    i ==Add        V:
      parse_Add()
    elseif    i ==Sub
   V:
      parse_Sub()

    …
    elseif    i ==Val
```