LL(1) Parsing, Refactoring and Recursive Descent

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
  - Assignment 4 is out and due June 14 (hopefully)
  - Midterm on Wednesday, June 19, in CHEB 170, 10 – 11:30am.
  - Final Exam on Friday, August 2, in CHEB 170, 1:00 – 4:00pm
- Readings:
  - Today: 2.3.0, 2.3.1
  - Note: I recommend using alternative texts for this part of the course:
  - E..g, Hopcorft et al, “Introduction to Automata Theory”
- Lecture Contents
  - 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 **a**, sentential form A . . ., either
  - $A \Rightarrow \alpha \Rightarrow^* \mathbf{a}\beta$
  - $A \Rightarrow \alpha \Rightarrow^* \varepsilon$ and derivation of A is succeeded by **a**.

- Intuitively, **a** is in the *predictor set* of $A \rightarrow \alpha$

  if $A\beta \Rightarrow \alpha\beta \Rightarrow^* a\gamma$, for $\beta,\gamma \in \Sigma^*$

  I.e., the parser selects $A \rightarrow \alpha$ if **a** is the input and in the *predictor set* of $A \rightarrow \alpha$

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 → + SS
  2. S → − SS
  3. S → ∗ SS
  4. S → / SS
  5. S → neg S
  6. S → integer

PREDICT Table

Production	Predictor Set
S → + S S	{+}
S → - S S	{-}
S → * S S	{*}
S → / S S	{/}
S → neg S	{neg}
S → integer	{integer}

Constructing PREDICT: The 3 Tables

- **FIRST(α):** the set of leftmost terminals **a** to be derived from $\alpha \in (V \cup \Sigma)^*$

  $\alpha \Rightarrow^* \mathbf{a}\beta$

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

  $S \Rightarrow^* \alpha X \mathbf{a}\beta$

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

The FIRST Table: Example

Grammar

- T → AB
- A → PQ
- A → BC
- P → pP
- P → ε
- Q → qQ
- Q → ε
- B → bB
- B → e
- C → cC
- C→ f

	Sym: σ	FIRST(σ)
Σ	p	{p}
	q	{q}
	b	{b}
	e	{e}
	c	{c}
	f	{f}
V	**T**	{p,q,b.e}
	A	{p,q,b.e,ϵ}
	P	{p,ϵ}
	Q	{q,ϵ}
	B	{b,e}
	C	{c,f}

The FIRST Table

- **Definition:** FIRST(σ), $\forall \sigma \in (V \cup \Sigma)^*$:
  - For $a \in \Sigma$, $a \in$ FIRST(σ) if $\sigma \Rightarrow^* a\beta$
  - $\varepsilon \in$ FIRST(σ) if $\sigma \Rightarrow^* \varepsilon$

Precompute FIRST sets for all terminals and variables

- **Notes:**
  - For $a \in \Sigma$, FIRST(a) = $\{a\}$
  - Precompute FIRST(X) only for $X \in V$
  - Generate FIRST(σ), $\sigma \in (V \cup \Sigma)^*$ as needed
- **To compute FIRST for terminals and variables**
  - For $a \in \Sigma$, FIRST(a) = $\{a\}$
  - For $X \in V$, FIRST(X) = $\emptyset$
  - Repeat until no new additions to FIRST(X), $X \in V$ are possible:

    $\forall (X \rightarrow \alpha) \in P$,

    FIRST(X) = FIRST(X) $\cup$ FIRST(α)

The FIRST Table (Part 2)

- **To compute FIRST(α) (in general)**
  - $\alpha = \alpha_1 \alpha_2 \ldots \alpha_k$, $\alpha_i \in V \cup \Sigma$
  - FIRST(α) = $\emptyset$
  - For i = 1,2...k:
    - FIRST(α) = FIRST(α) $\cup$ (FIRST(α_i)$\setminus\{\varepsilon\}$)
    - if $\varepsilon \notin$ FIRST(α_i) then return
  - FIRST(α) = FIRST(α) $\cup$ ε

The FIRST Table: Example

Loop until no no more changes:
$\forall X \rightarrow \alpha \in P$,
FIRST(X) = FIRST(X) ∪ FIRST(α)

Grammar

- T → AB
- A → PQ
- A → BC
- P → pP
- P → ε
- Q → qQ
- Q → ε
- B → bB
- B → e
- C → cC
- C→ f

	Symbol	Iter. 0	Iter. 1	Iter.2	Iter. 3
Σ	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}
V	**T**	Ø	Ø	Ø	{p,q,b.e}
	A	Ø	Ø	{p,q,b.e,ϵ}	{p,q,b.e,ϵ}
	P	Ø	{p,ϵ}	{p,ϵ}	{p,ϵ}
	Q	Ø	{q,ϵ}	{q,ϵ}	{q,ϵ}
	B	Ø	{b,e}	{b,e}	{b,e}
	C	Ø	{c,f}	{c,f}	{c,f}

The FOLLOW Table

- **Definition:** FOLLOW(X), $\forall$ X $\in$ V:
  - For a $\in \Sigma$, a $\in$ FOLLOW(X) if $\exists$ S $\Rightarrow^*$ α**Xa**β
  - $\varepsilon \in$ FOLLOW(X) if S $\Rightarrow^*$ αX
- **To Compute FOLLOW**
  - FOLLOW(S) = {ε}
  - For X $\in$ V, FOLLOW(X) = $\emptyset$
  - Repeat until no new additions to FOLLOW(X), X $\in$ V are possible:
    - For each B $\in$ V
      - $\forall$ (X $\rightarrow$ αBβ) $\in$ P,
        - FOLLOW(B) = FOLLOW(B) $\cup$ (FIRST(β)\\{ε})
        - if $\varepsilon \in$ FIRST(β) then FOLLOW(B) = FOLLOW(B) $\cup$ FOLLOW(X)

The FOLLOW Table: Example

Grammar

- $T \rightarrow AB$
- $A \rightarrow PQ$
- $A \rightarrow BC$
- $P \rightarrow pP$
- $P \rightarrow \varepsilon$
- $Q \rightarrow qQ$
- $Q \rightarrow \varepsilon$
- $B \rightarrow bB$
- $B \rightarrow e$
- $C \rightarrow cC$
- $C \rightarrow f$

Symbol	Iter. 0	Iter. 1	Iter.2
T	{ϵ}	{ϵ}	{ϵ}
A	∅	{b,e}	{b,e}
P	∅	{q}	{q,b,e}
Q	∅	∅	{b,e}
B	∅	{ϵ,c,f}	{ϵ,c,f}
C	∅	∅	{b,e}

FOLLOW(S) = {ε}
For X ∈ V, FOLLOW(X) = ∅
Repeat until no new additions to FOLLOW(X), X ∈ V are possible:
    For each B ∈ V
        ∀X → αBβ ∈ P,
            FOLLOW(B) = FOLLOW(B) ∪ (FIRST(β)\{ε})
            if ε ∈ FIRST(β) then FOLLOW(B) = FOLLOW(B) ∪ FOLLOW(X)

The PREDICT Table

- **Definition:** For $a \in \Sigma \cup \{\varepsilon\}$, $a \in \text{PREDICT}(A \rightarrow \alpha)$ if
  - $a \in \text{FIRST}(\alpha) \setminus \{\varepsilon\}$ or
  - $\varepsilon \in \text{FIRST}(\alpha)$ and $a \in \text{FOLLOW}(A)$
- **To Compute PREDICT**
  - For each $(A \rightarrow \alpha) \in P$, $\text{PREDICT}(A \rightarrow \alpha) = \emptyset$
  - For each $(A \rightarrow \alpha) \in P$
    - $\text{PREDICT}(A \rightarrow \alpha) = \text{FIRST}(\alpha) \setminus \{\varepsilon\}$
    - if $\varepsilon \in \text{FIRST}(\alpha)$ then

      $\text{PREDICT}(A \rightarrow \alpha) = \text{PREDICT}(A \rightarrow \alpha) \cup \text{FOLLOW}(A)$

The PREDICT Table: Example

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

For each (A→α) ∈ P
  PREDICT(A→α) = FIRST(α) \ {ε}
  if ε ∈ FIRST(α) then
    PREDICT(A→α) = PREDICT(A→α) ∪ 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 → ϵ	{q,b,e}
Q → qQ	{q}
Q → ϵ	{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 for the grammar
- This grammar is not LL(1) if and only If 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 languauges 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

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

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta A'$$
$$A' \rightarrow \varepsilon$$

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

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

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

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

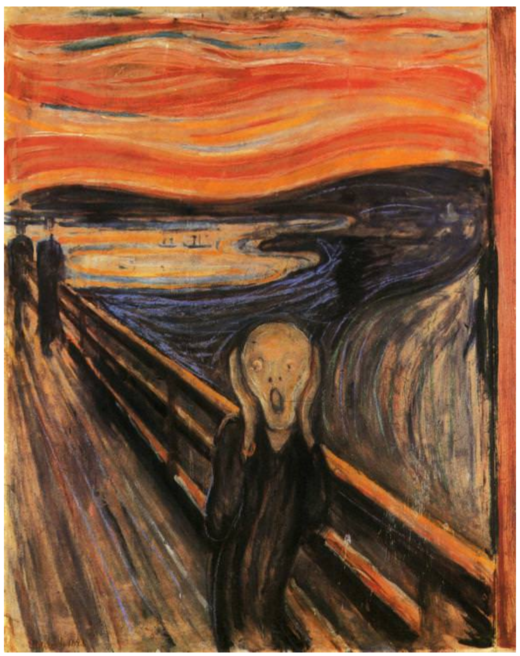
- Two efficient approaches:
  - Recursive Descent
  - Deterministic Pushdown Automata (DPDA)

- Recursive Descent is easier to understand and implement.

Recursive Descent

Use PREDICT table

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



```
parse_X:
  t = peek_next_token()
  select X → α based on t
  for each αi ∈ α :
    if αi == Y1 ∈ V:
      parse_Y1()
    elseif αi == Y2 ∈ V:
      parse_Y2()

    …
    elseif αi == Yk ∈ V:
      parse_Yk()
    elseif αi == t:
      remove_token()
      t = peek_next_token()
    else:
      syntax error
```

Example

Grammar

- S → Add | Sub | Mul
- S → Div | Neg | Val
- Add → + S S
- Sub → – S S
- Mul → ∗ S S
- Div → / S S
- Neg → neg S
- Val → integer

```
parse_S:
  t = peek_at_token()
  select S → α based on t
  for each αi ∈ α :
    if αi == Add ∈ V:
      parse_Add()
    elseif αi == Sub ∈ V:
      parse_Sub()

    …
    elseif αi == Val ∈ V:
      parse_Val()
    elseif αi == t:
      remove_token()
      t = peek_at_token()
    else:
      syntax error
```

Example
More Concrete

Grammar

- **S → Add | Sub | Mul**
- **S → Div | Neg | Val**
- Add → + S S
- Sub → – S S
- Mul → ∗ S S
- Div → / S S
- Neg → neg S
- Val → integer

```
parse_S:
  t = peek_at_token()
  if t == '+' :
    n = parse_Add()
  elseif t == '-':
    n = parse_Sub()
  elseif t == '*':
    n = parse_Mul()
  elseif t == '/':
    n = parse_Div()
  elseif t == 'neg':
    n = parse_Neg()
  elseif t is integer:
    n = parse_Val()
  else:
    syntax error

  return TreeNode(n)
```

Example
More Concrete

Grammar

- S → Add | Sub | Mul
- S → Div | Neg | Val
- **Add → + S S**
- Sub → – S S
- Mul → ∗ S S
- Div → / S S
- Neg → neg S
- Val → integer

```
parse_Add:
  t = peek_at_token()
  if t != '+':
    # never happens
    syntax error
  remove_token()
  s1 = parse_S()
  s2 = parse_S()
  return TreeNode(t,s1,s2)
```