CS 4240: Compilers

Lecture 22: LL(1) Parsing

Instructor: Vivek Sarkar
(vsarkar@gatech.edu)

April 10, 2019

ANNOUNCEMENTS & REMINDERS

- **Project 3 assigned on April 8th**
  - **Due by 11:59pm on Tuesday, April 23rd**
  - **Automatic penalty-free extension until 11:59pm on Tuesday, April 30th**
  - 10% of course grade
- **Homework 3 assigned today**
  - **Due by 11:59pm on Tuesday, April 23rd**
  - **Automatic penalty-free extension until 11:59pm on Friday, April 26th**
  - 5% of course grade
- **FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm**
  - 30% of course grade

Worksheet # 21 Solution

(From Lecture #21 given on 4/8/2019)

1. Consider the following left recursive grammar.

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \epsilon$$

Convert the above grammar to one that is suitable for LL parsing by removing left recursion from it.

- **The grammar can be represented as a 4-tuple. G = (S, NT, T, P)**

S is the start symbol
NT = **{S, A}** (Set of non-terminal variables)
T = **{a, b, c, d}** (Set of terminal variables)
P = **{S ⇒ Aa, S ⇒ b, A ⇒ Ac, A ⇒ Sd, A ⇒ ε}** (Set of productions)

1. Consider the following left recursive grammar.

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \epsilon$$

Convert the above grammar to one that is suitable for LL parsing by removing left recursion from it.

- **A grammar is left recursive if ∃ A ∈ NT such that ∃ a derivation A ⇒+ Aα, for some string α ∈ (NT ∪ T)+**

- **S** ⇒ Aa ⇒ **S**da (indirect left recursion)
- **A** ⇒ **A**c (direct left recursion)

Algorithm to remove left recursion

- The algorithm removes left recursion via 2 techniques.
  1. **convert <u>indirect left recursion</u> to <u>direct left recursion</u>**
  2. **Rewrite <u>direct left recursion</u> as <u>right recursion</u>**

```
impose an order on the nonterminals, A_1, A_2, ..., A_n

for i ← 1 to n do;
    for j ← 1 to i - 1 do;
        if ∃ a production A_i → A_j γ
            then replace A_i → A_j γ with one or more
                productions that expand A_j
    end;
    rewrite the productions to eliminate
        any direct left recursion on A_i
end;
```

1

2

Using the algorithm : 1st iteration of outer loop

- Let's impose the order of non-terminals : **S, A**

```
impose an order on the nonterminals, A1, A2, ..., An

for i ← 1 to n do;
    for j ← 1 to i - 1 do;
        if ∃ a production Ai→Ajγ
            then replace Ai→Ajγ with one or more
                productions that expand Aj
    end;
    rewrite the productions to eliminate
        any direct left recursion on Ai
end;
```

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \epsilon$$

Nothing is done in the inner loop.
No direct recursion on S exists.
No change is made to the grammar.

Using the algorithm : 2nd iteration of outer loop

- Imposed order of non-terminals : **S, A**

```
impose an order on the nonterminals, A1, A2, ..., An

for i ← 1 to n do;
    for j ← 1 to i - 1 do;
        if ∃ a production Ai → Aj γ
            then replace Ai → Aj γ with one or more
                productions that expand Aj
    end;
    rewrite the productions to eliminate
        any direct left recursion on Ai
end;
```

S ⇒ Aa | b

A ⇒Ac | **S**d | ε

In the inner loop, production **A ⇒ Sd** is replaced with **A ⇒Aad, A ⇒bd.**

S ⇒ Aa | b

A ⇒ Ac | Aad | bd | ε

Using the algorithm : 2nd iteration of outer loop

- Imposed order of non-terminals : **A_1 = S, A_2 = A**

```
impose an order on the nonterminals, A1, A2, ..., An

for i ← 1 to n do;
    for j ← 1 to i - 1 do;
        if ∃ a production Ai→Ajγ
            then replace Ai→Ajγ with one or more
                productions that expand Aj
    end;
    rewrite the productions to eliminate
        any direct left recursion on Ai
end;
```

S ⇒ Aa | b

A ⇒ Ac | Aad |

bd | ε

We have to rewrite <u>direct left recursion</u> as <u>right recursion.</u>

Removing direct left recursion in the grammar

```
A_productions = [ A ⇒ Ac , A ⇒ Aad, A ⇒ bd, A ⇒ ε]
Aprime_productions = []
for production in A_productions:
    if production result starts with A:
        move A in the production result to end of result
        popped = A_productions.pop(production)
        replace every A in popped to A'
        A_prime_productions.push(popped)
    else:
        append A' to the production result
Aprime_productions.push(A' ⇒ ε)
```

S ⇒ Aa | b

A ⇒ Ac | Aad | bd | ε

- **Resulting grammar (right-recursive grammar)**

  S ⇒ Aa | b

  A ⇒ bdA’ | A’

  A’ ⇒ cA’ | adA’ | ε

Page 27 of slides from lecture21, covers on how to remove direct left recursion in a grammar.

Roadmap (Where are we?)

We set out to study parsing

» Specifying syntax
  – Context-free grammars ✓
  – Ambiguity ✓

» Top-down parsers
  – Algorithm & its problem with left recursion ✓
  – Left-recursion removal ✓

» Predictive top-down parsing
  – The LL(1) condition
  – Simple recursive descent parsers
  – Table-driven LL(1) parsers

Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between α & β

FIRST sets

For some rhs $\alpha \in G$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) iff $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

The LL(1) Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

FIRST(α) $\cap$ FIRST(β) = $\varnothing$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol !

This is almost correct
See the next slide

Predictive Parsing

What about ε-productions?

⇒ They complicate the definition of LL(1)

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in \text{FIRST}(\alpha)$, then we need to ensure that $\text{FIRST}(\beta)$ is disjoint from $\text{FOLLOW}(A)$, too, where

$\text{FOLLOW}(A)$ = the set of terminal symbols that can immediately follow A in a sentential form

Define $\text{FIRST}^+(A \rightarrow \alpha)$ as

» $\text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$, if $\varepsilon \in \text{FIRST}(\alpha)$

» $\text{FIRST}(\alpha)$, otherwise

Then, a grammar is LL(1) iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies

$$\text{FIRST}^+(A \rightarrow \alpha) \cap \text{FIRST}^+(A \rightarrow \beta) = \varnothing$$

Predictive Parsing

Given a grammar that has the LL(1) property

- Can write a simple routine to recognize each lhs
- Code is both simple & fast

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$$\text{FIRST}^+(A\rightarrow\beta_i) \cap \text{FIRST}^+(A\rightarrow\beta_j) = \emptyset \text{ if } i \neq j$$

```
/* find an A */
if (current_word ∈ FIRST(A→β1))
   find a β1 and return true
else if (current_word ∈ FIRST(A→β2))
   find a β2 and return true
else if (current_word ∈ FIRST(A→β3))
   find a β3 and return true
else
   report an error and return false
```

Grammars with the LL(1) property are called predictive grammars because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the LL(1) property are called predictive parsers.

One kind of predictive parser is the recursive descent parser.

Recursive Descent Parsing

Recall the expression grammar, after transformation

1	*Goal*	$\rightarrow$	*Expr*
2	*Expr*	$\rightarrow$	*Term Expr′*
3	*Expr′*	$\rightarrow$	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	$\rightarrow$	*Factor Term′*
7	*Term′*	$\rightarrow$	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	$\rightarrow$	<u>number</u>
11		|	<u>id</u>
12		|	<u>(</u> Expr <u>)</u>

This produces a parser with six <u>mutually recursive</u> routines:

- Goal
- Expr
- EPrime
- Term
- TPrime
- Factor

Each recognizes one NT or T

The term <u>descent</u> refers to the direction in which the parse tree is built.

Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

```
Goal( )
   token ← next_token( );
   if (Expr( ) = true & token = EOF)
      then next compilation step;
      else
         report syntax error;
         return false;

Expr( )
  if (Term( ) = false)
     then return false;
     else return Eprime( );
```

```
Factor( )
  if (token = Number) then
     token ← next_token( );
     return true;
  else if (token = Identifier) then
     token ← next_token( );
     return true;
  else if (token = Lparen)
     token ← next_token( );
     if (Expr( ) = true & token = Rparen) then
        token ← next_token( );
        return true;
  // fall out of if statement
  report syntax error;
     return false;
```

looking for Number, Identifier, or "(", found token instead, or failed to find Expr or ")" after "("

***EPrime*, *Term*, & *TPrime* follow the same basic lines**

Recursive Descent (Summary)

1. Massage grammar to have LL(1) condition
   a. Remove left recursion
   b. Build FIRST (and FOLLOW) sets
   c. Left factor it, as needed
2. Define a procedure for each non-terminal
   a. Implement a case for each right-hand side
   b. Call procedures as needed for non-terminals
3. Add extra code, as needed
   a. Perform context-sensitive checking
   b. Build an IR to record the code

Can we automate this process?

Roadmap (Where are we?)

We set out to study parsing

- Specifying syntax
  - Context-free grammars ✓
  - Ambiguity ✓
- Top-down parsers
  - Algorithm & its problem with left recursion ✓
  - Left-recursion removal ✓
- Predictive top-down parsing ✓
  - The LL(1) condition ✓
  - Simple recursive descent parsers ✓
  - First and Follow sets
  - Table-driven LL(1) parsers

FIRST and FOLLOW Sets

FIRST(α)

For some $\alpha \in (T \cup NT)^*$, define FIRST(α) as the set of tokens that appear as the first symbol in some string that derives from α

That is, $\underline{x} \in$ FIRST(α) iff $\alpha \Rightarrow^* \underline{x}\,\gamma$, for some γ

FOLLOW(A)

For some $A \in NT$, define FOLLOW(A) as the set of symbols that can occur immediately after A in a valid sentential form

FOLLOW(S) = {EOF}, where S is the start symbol

To build FOLLOW sets, we need FIRST sets ...

Computing FIRST Sets

```
for each x ∈ T, FIRST(x) ← { x }
for each A ∈ NT, FIRST(A) ← Ø

while (FIRST sets are still changing) do
   for each p ∈ P, of the form A→β do
      if β is B1B2…Bk then begin;
         rhs ← FIRST(B1) - { ε }
         for i ← 1 to k-1 by 1 while ε ∈ FIRST(Bi) do
             rhs ← rhs ∪ ( FIRST(Bi+1) - { ε } )
             end  // for loop
      end        // if-then
       if  i = k and ε ∈ FIRST(Bk )
          then rhs ← rhs ∪ { ε }
        FIRST(A) ← FIRST(A) ∪ rhs
       end  // for loop
   end      // while loop
```

Outer loop is monotone increasing for FIRST sets

→ | T ∪ NT ∪ ε | is bounded, so it terminates

Inner loop is bounded by the length of the productions in the grammar

Computing FOLLOW Sets

```
for each A ∈ NT, FOLLOW(A) ← Ø

FOLLOW(S) ← {EOF }

while (FOLLOW sets are still changing)
    for each p ∈ P, of the form A→B1B2 … Bk
        TRAILER ← FOLLOW(A)
        for i ← k down to 1
            if Bi ∈ NT then                             // domain check
                FOLLOW(Bi) ← FOLLOW(Bi) ∪ TRAILER
                if ε ∈ FIRST(Bi )                       // add right context
                    then TRAILER ← TRAILER ∪ ( FIRST(Bi) – { ε } )
                    else TRAILER ← FIRST(Bi)   // no ε => no right context
            else TRAILER ← {Bi }                        // Bi ∈ T => only 1 symbol
```

Example of FIRST and FOLLOW sets

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr'*
3	*Expr'*	→	+ *Term Expr'*
4		|	- *Term Expr'*
5		|	ε
6	*Term*	→	*Factor Term'*
7	*Term'*	→	* *Factor Term'*
8		|	/ *Factor Term'*
9		|	ε
10	*Factor*	→	number
11		|	id
12		|	(*Expr*)

FIRST+(A→β) is identical to FIRST(β) except for productions 5 and 9

FIRST+(Expr'→ ε) is {ε,), eof}

FIRST+(Term'→ ε) is {ε,+,-,), eof}

Symbol	FIRST	FOLLOW
num	num	
name	name	
+	+	
-	-	
*	*	
/	/	
(	(	
)	)	
eof	eof	
ε	ε	
Goal	(,name,num	eof
Expr	(,name,num	), eof
Expr'	+, -, ε	), eof
Term	(,name,num	+, -,), eof
Term'	*, /, ε	+,-,), eof
Factor	(,name,num	+, -, *, /,), eof

Example of FIRST^{+} sets

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	number
11		|	id
12		|	(*Expr*)

Prod'n	FIRST+
1	(,name,num
2	(,name,num
3	+
4	-
5	ε,), eof
6	(,name,num
7	*
8	/
9	ε,+,-,), eof
10	number
11	name
12	(

Building Top-down Parsers

Strategy

- Encode knowledge in a table
- Use a standard "skeleton" parser to interpret the table

Example

- The non-terminal Factor has 3 expansions
  - (Expr) or Identifier or Number
- Table might look like:

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	number
11		|	id
12		|	(*Expr*)

Terminal Symbols

Non-terminal Symbols

	+	-	*	/	Num	Id	(	)	EOF
Factor	—	—	—	—	10	11	12	—	—

Error on `+'

Reduce by rule 10 on Num

Building Top Down Parsers

Building the complete table

- Need an interpreter for the table (skeleton parser)
- Need a row for every NT & a column for every T

LL(1) Skeleton Parser

```
word ← NextWord()               // Initial conditions, including
push EOF onto Stack              // a stack to track local goals
push the start symbol, S, onto Stack
TOS ← top of Stack

loop forever
  if TOS = EOF and word = EOF then
     break & report success     // exit on success

   else if TOS is a terminal then
     if TOS matches word then
        pop Stack               // recognized TOS
        word ← NextWord()
     else report error looking for TOS  // error exit

   else                          // TOS is a non-terminal
     if TABLE[TOS,word] is A→ B1B2…Bk then
        pop Stack               // get rid of A
        push Bk, Bk-1, …, B1    // in that order
     else break & report error expanding TOS
  TOS ← top of Stack
```

Building Top Down Parsers

Building the complete table

- Need a row for every NT & a column for every T
- Need a table-driven interpreter for the table
- Need an algorithm to build the table

Filling in TABLE[X,y], $X \in NT$, $y \in T$

1. entry is the rule $X \rightarrow \beta$, if $y \in FIRST^{+}(X \rightarrow \beta)$
- entry is error if rule 1 does not define

If any entry has more than one rule, G is not LL(1)

This is the LL(1) table construction algorithm

Example of LL(1) Parsing Table (entry = production #)

	+	-	*	/	Num	Id	(	)	EOF
Goal	—	—	—	—	1	1	1	—	—
Expr	—	—	—	—	2	2	2	—	—
Expr'	3	4	—	—	—	—	—	5	5
Term	—	—	—	—	6	6	6	—	—
Term'	9	9	7	8	—	—	—	9	9
Factor	—	—	—	—	10	11	12	—	—