Parsing IIIa

(Top-down parsing: recursive descent & *LL(1)*)

COMP 412
Fall 2005

Copyright 2005, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

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 today
  - Table-driven LL(1) parsers today

Recursive Descent (Summary)

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

Can we automate this process?

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 FIRST sets, we need FOLLOW sets …

Computing FIRST Sets

for each $x \in T$, *FIRST(x)* $\leftarrow$ { x }
for each $A \in NT$, *FIRST(A)* $\leftarrow \varnothing$

while (FIRST sets are still changing)
    for each $p \in P$, *of the form* $A \rightarrow \beta$,
        if β *is* ε *then*
            FIRST(A) $\leftarrow$ *FIRST(A)* $\cup$ { ε }
        else // $\beta = B_1 B_2 \ldots B_k$
            FIRST(A) $\leftarrow$ *FIRST(A)* $\cup$ (*FIRST*(B_1) – { ε })
            for $i \leftarrow 1$ *to* $k-1$ *by* 1 *while* $\varepsilon \in$ *FIRST*(B_i)
                FIRST(A) $\leftarrow$ *FIRST(A)* $\cup$ (*FIRST*(B_{i+1}) – { ε })
            if $i = k$ *and* $\varepsilon \in$ *FIRST*(B_k)
                then FIRST(A) $\leftarrow$ *FIRST(A)* $\cup$ { ε }

clumsy test on exit condition of loop

For *SheepNoise:*

FIRST(*Goal*) = { baa }
FIRST(*SN*) = { baa }
FIRST(baa) = { baa }

Computing FIRST Sets

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

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

- Outer loop is monotone increasing for FIRST sets
  - → $|\ T \cup NT \cup \varepsilon\ |$ is bounded, so it terminates
- Inner loop is bounded by length of productions in grammar

Computing FOLLOW Sets

```
FOLLOW(S) ← {EOF }
for each A ∈ NT, FOLLOW(A) ← Ø

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 ) then             // add right context
                    TRAILER ← TRAILER ∪ ( FIRST(Bi) – { ε } )
                else TRAILER ← FIRST(Bi)           // no ε => no right context
            else TRAILER ← {Bi }                   // Bi ∈ T => only 1 symbol
```

For *SheepNoise:*

FOLLOW(*Goal*) = { EOF }

FOLLOW(*SN*) = { baa, EOF }

Building Top-down Parsers

Given an *LL(1)* grammar, and its FIRST & FOLLOW sets …

- Emit a routine for each non-terminal
  - Nest of if-then-else statements to check alternate rhs's
  - Each returns true on success and throws an error on false
  - Simple, working *(, perhaps ugly,)* code
- This automatically constructs a recursive-descent parser

I don't know of a system that does this …

Improving matters

- Nest of if-then-else statements may be slow
  - Good case statement implementation would be better
- What about a table to encode the options?
  - Interpret the table with a skeleton, as we did in scanning

Building Top-down Parsers

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*	→	id
11		|	number
12		|	(Expr)

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:

Terminal Symbols

Non-terminal Symbols	+	-	*	/	Id.	Num.	(	)	EOF
Factor	—	—	—	—	10	11	12	—	—

Error on `+'

Reduce by rule 10 on `id'

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

LL(1) Skeleton Parser

```
token ← next_token()                 // 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 token = EOF then
     break & report success  ◄──────── exit on success
  else if TOS is a terminal then
     if TOS matches token then
        pop Stack                    // recognized TOS
        token ← next_token()
     else report error looking for TOS
  else                               // TOS is a non-terminal
     if TABLE[TOS,token] 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  ◄──── exit on failure
  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$^+(X \rightarrow \beta)$
2. 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

LL(1) Expression Parser

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*	→	id
11		|	number
12		|	(Expr)

FIRST(*Goal*) = FIRST(*Expr*) =
FIRST(*Term*) = FIRST(*Factor*) =
{ id, number, (}

FIRST(*Expr′*) = { +, -, ε }

FIRST(*Term′*) = { *, /, ε }

FOLLOW(*Goal*) = { EOF }

FOLLOW(*Expr*) = {), EOF }

FOLLOW(*Expr′*) = {), EOF }

FOLLOW(*Term*) = { +, -,), EOF }

FOLLOW(*Term′*) = { +, -,), EOF }

FOLLOW(*Factor*) =
{ +, -, *, /,), EOF }

LL(1) Expression Parsing Table

	+	-	*	/	Id	Num	(	)	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	—	—

Recursive Descent in OO Languages

- Shortcomings of Recursive Descent
  - Too procedural
  - No convenient way to build parse tree
- Solution
  - Associate a class with each non-terminal symbol
    - Allocated object contains pointer to the parse tree

```
Class NonTerminal {
public:
        NonTerminal(Scanner & scnr) { s = &scnr; tree = NULL; }
        virtual ~NonTerminal() { }
        virtual bool isPresent() = 0;
        TreeNode * abSynTree() { return tree; }
protected:
        Scanner * s;
        TreeNode * tree;
}
```

Non-terminal Classes

```
Class Expr : public NonTerminal {
public:
        Expr(Scanner & scnr) : NonTerminal(scnr) { }
        virtual bool isPresent();
}

Class EPrime : public NonTerminal {
public:
        EPrime(Scanner & scnr, TreeNode * p) :
                NonTerminal(scnr) { exprSofar = p; }
        virtual bool isPresent();
protected:
        TreeNode * exprSofar;
}

…  // definitions for Term and TPrime

Class Factor : public NonTerminal {
public:
        Factor(Scanner & scnr) : NonTerminal(scnr) { };
        virtual bool isPresent();
}
```

Implementation of isPresent

```
bool Expr::isPresent() {  // Expr -> Term Expr'

        Term * operand1 = new Term(*s);   // instantiate Term & check for its presence
        if (!operand1->isPresent()) return FALSE;

        Eprime * operand2 = new EPrime(*s, NULL);  // instantiate Eprime & check

        if (!operand2->isPresent()) return TRUE;
        else return FALSE;

}
```

Implementation of isPresent

```
bool EPrime::isPresent() {   // Expr' -> + Term Expr' | - Term Expr' | ε

    token_type op = s->nextToken();
    if (op == PLUS || op == MINUS) {      // check for presence of + or =
        s->advance();

        Term * operand2 = new Term(*s);   // instantiate Term & check it
        if (!operand2->isPresent()) throw SyntaxError(*s);

        Eprime * operand3 = new EPrime(*s, NULL);  // instantiate & check EPrime

        if (operand3->isPresent()) return TRUE;
        else return FALSE

    }
    else return TRUE; // corresponds to EPRIME → ε

}
```

Abstract Syntax Tree Construction

```
bool Expr::isPresent() { // with code to build a parse tree

        Term * operand1 = new Term(*s);
        if (!operand1->isPresent()) return FALSE;
        tree = operand1->abSynTree();

        EPrime * operand2 = new EPrime(*s, tree);
        if (operand2->isPresent()) {
                tree  = operand2->absSynTree();
                return TRUE;
        }
        else return FALSE

        // upon return, tree is either the tree for the Term
        //              or the tree for Term followed by EPrime
}
```

Abstract Syntax Tree Construction

```
bool EPrime::isPresent() { // with code to build a parse tree
    token_type op = s->nextToken();
    if (op == PLUS || op == MINUS) {
        s->advance();

        Term * operand2 = new Term(*s);
        if (!operand2->isPresent()) throw SyntaxError(*s);

        TreeNode * t2 = operand2->absSynTree();
        tree = new TreeNode(op, exprSofar, t2);

        Eprime * operand3 = new Eprime(*s, tree);
        if (operand3->isPresent()) {
            tree = operand3->absSynTree();
            return TRUE;
        }
        else return FALSE;
    }
    else return TRUE; // corresponds to EPRIME → ε
}
```

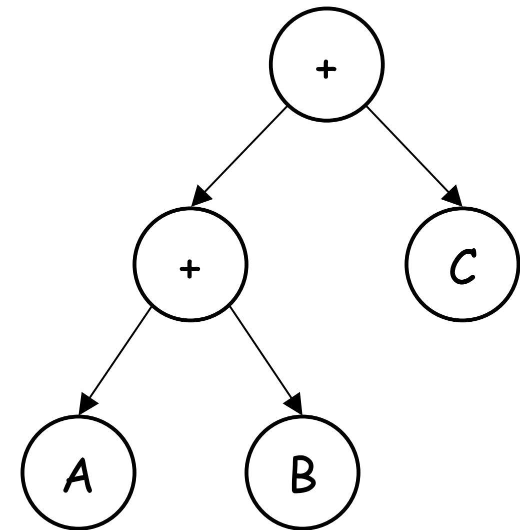
Factor

```
bool Factor::isPresent() { // with code to build a parse tree
    token_type op = s->nextToken();

    if (op == IDENTIFIER | op == NUMBER) {
        tree = new TreeNode(op, s->tokenValue());
        s->advance();
        return TRUE;
    }
    else if (op == LPAREN) {
        s->advance();
        Expr * operand = new Expr(*s);
        if (!operand->isPresent()) throw SyntaxError(*s);
        if (s->nextToken() != RPAREN) throw SyntaxError(*s);
        s->advance();
        tree = operand->absSynTree();
        return TRUE;
    }
    else return FALSE;
}
```

OO Recursive Descent Example: A+B+C

- Instantiate Expr
  - Instantiate Term, Factor
    - Factor returns True, tree = A
    - Term returns True, tree = A
  - Instantiate Eprime (exprsofar = A)
    - Match +, Instantiate Term
    - Term returns True, tree = B
    - Eprime tree = A + B
  - Instantiate Eprime (exprsofar = {A + B})
    - Match +, Instatiate Term
    - Term returns tree = C
    - Eprime tree = {A+B}+C
  - Instantiate Eprime (exprsofar ={{A + B} + C})
    - Return true with no new tree



Extra Slides Start Here

Computing FIRST Set for a String

if β *is* ε *then*

    FIRST(β) ← { ε }

else

    let β *be* $B_1B_2 \ldots B_k$

    FIRST(β) ← *FIRST*(B_1) – { ε }

    for i ← *1 to k–1 by 1 while* $\varepsilon \in$ *FIR*[illegible]

        FIRST(β) ← *FIRST*(β) ∪ (*F*[illegible] ε })

    if i = k and $\varepsilon \in$ *FIRST*(*B*[illegible]

        then FIRST(β) ← [illegible] ε }

Since EaC, and the slides, use the augmented FIRST set, FIRST+, I elided this slide and moved it to the back of the set…

FIRST is defined on symbols, not strings

- LL(1) construction needs to apply it to an entire RHS
- Either extend FIRST to strings or introduce FIRST+ set