Parsing — Part II

(Top-down parsing, left-recursion removal)

Parsing Techniques

Top-down parsers *(LL(1), recursive descent)*

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers *(LR(1), operator precedence)*

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Top-down Parsing

A top-down parser starts with the root of the parse tree

The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:

Construct the root node of the parse tree

Repeat until the fringe of the parse tree matches the input string

1. *At a node labeled A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child*
2. *When a terminal symbol is added to the fringe and it doesn't match the fringe, backtrack*
3. *Find the next node to be expanded* *(label $\in$ NT)*

- The key is picking the right production in step 1
  - → *That choice should be guided by the input string*

Remember the expression grammar?

Version with precedence derived last lecture

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id

And the input x - 2 * y

A possible parse

Consider the following parse of x - 2 * y

Rule	*Sentential Form*	*Input*
—	*Goal*	↑x - 2 * y
1	*Expr*	↑x - 2 * y
2	*Expr + Term*	↑x - 2 * y
2	*Expr + Term + Term*	↑x - 2 * y
2	*Expr+ Term + Term + Term*	↑x - 2 * y
2	*Expr+Term + Term + …+ Term*	↑x - 2 * y

consuming no input !

This doesn't terminate *(obviously)*

- Wrong choice of expansion leads to non-termination
- Non-termination is a bad property for a parser to have
- Parser must make the right choice

Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is *left recursive* if $\exists\ A \in NT$ such that
$\exists$ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$

Our expression grammar is left recursive

- This can lead to non-termination in a top-down parser
- For a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Non-termination is a bad property in any part of a compiler

Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

$$Fee \rightarrow Fee\ \alpha$$
$$\quad\quad |\ \beta$$

where neither α nor β start with *Fee*

We can rewrite this as

$$Fee \rightarrow \beta\ Fie$$
$$Fie \rightarrow \alpha\ Fie$$
$$\quad\quad |\ \varepsilon$$

where *Fie* is a new non-terminal

This accepts the same language, but uses only right recursion

Picking the "Right" Production

If it picks the wrong production, a top-down parser may backtrack

Alternative is to look ahead in input & use context to pick correctly

How much lookahead is needed?

- In general, an arbitrarily large amount

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are *LL(1)* and *LR(1)* grammars

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

We will defer the problem of how to compute FIRST sets until we look at the *LR(1)* table construction algorithm

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

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$$

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 $\varepsilon \in$ FIRST(α), then we need to ensure that FIRST(β) is disjoint from FOLLOW(α), too

FOLLOW(α) is the set of all words in the grammar that can legally appear immediately after an α

Define FIRST$^+$(α) as

- FIRST(α) $\cup$ FOLLOW(α), if $\varepsilon \in$ FIRST(α)
- FIRST(α), otherwise

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

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

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

$$\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \varnothing$$

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.

Recursive Descent Parser

We build a recursive descent parser for the following grammar:

$$A \rightarrow B \mid CA \mid a$$
$$B \rightarrow bB \mid x$$
$$C \rightarrow c$$

The term descent refers to the direction in which the parse tree is built.

Recursive Descent Parsing

To actually build a parse tree:

- Augment parsing routines to build nodes
- Node for each symbol on *rhs*
- Action is to receive all *rhs* nodes, make them children of *lhs* node, and return this new node

```
B( )
        if (lookahead( ) = b)
    then return new BNode(read( ),B( ));
if (lookahead( ) = x)
    then return new BNode(read( ));
throw Exception;
```

To build an abstract syntax tree

- Build fewer nodes
- Put them together in a different order

```
B( )
        if (lookahead( ) = b)
    then return B( ).addOne( );
if (lookahead( ) = x)
    then return new BNode(0);
throw Exception;
```

This is a preview of Chapter 4

Left Factoring

What if my grammar does not have the LL(1) property?

⇒ Sometimes, we can transform the grammar

How would you rewrite the grammar

$$A \rightarrow aab \mid aac \mid aad$$

?

Left Factoring

What if my grammar does not have the LL(1) property?

$\Rightarrow$ Sometimes, we can transform the grammar

How would you rewrite the grammar

$$A \rightarrow aab \mid aac \mid aad$$

Rewrite to

$$A \rightarrow aa\, A'$$

$$A' \rightarrow b \mid c \mid d$$

Left Factoring (Generality)

Question

By *eliminating left recursion* and *left factoring*, can we transform an arbitrary *CFG* to a form where it meets the *LL(1)* condition? (and can be parsed predictively with a single token lookahead?)

Answer

Given a *CFG* that doesn't meet the *LL(1)* condition, it is undecidable whether or not an equivalent *LL(1)* grammar exists.

Example

$\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

Language that Cannot Be LL(1)

Example

$\{a^n 0\, b^n \mid n \geq 1\} \cup \{a^n 1\, b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

$G \rightarrow \underline{a}A\underline{b}$

$\quad\mid \underline{a}B\underline{bb}$

$A \rightarrow \underline{a}A\underline{b}$

$\quad\mid \underline{0}$

$B \rightarrow \underline{a}B\underline{bb}$

$\quad\mid \underline{1}$

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group.

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?

Building Top-down Parsers

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

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

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

Improving matters

- Bunch of if-then 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

Strategy

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

Example

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

Terminal Symbols

Non-terminal Symbols

	+	-	*	/	Id.	Num.	EOF
Factor	—	—	—	—	10	11	—

Error on `+'

Reduce by rule 10 on `+'

Building Top Down Parsers

Building the complete table

- Need a row for every *NT* & a column for every *T*
- 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(β)
2. entry is the rule $X \rightarrow \varepsilon$ if y $\in$ FOLLOW(X) and $X \rightarrow \varepsilon \in G$
3. entry is error if neither 1 nor 2 define it

If any entry is defined multiple times, *G* is not *LL(1)*

This is the *LL(1)* table construction algorithm

LL(1) Skeleton Parser

```
word ← nextWord()
push EOF onto Stack
push the start symbol onto Stack
TOS ← top of Stack
loop forever
  if TOS = EOF and word = EOF then          ← exit on success
      report success and exit
  else if TOS is a terminal or eof then
    if TOS matches word then
       pop Stack                            // recognized TOS
       word ← nextWord()
    else
       report error looking for TOS
  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 on stack        // in that order
    else report error expanding TOS
  TOS ← top of Stack
```