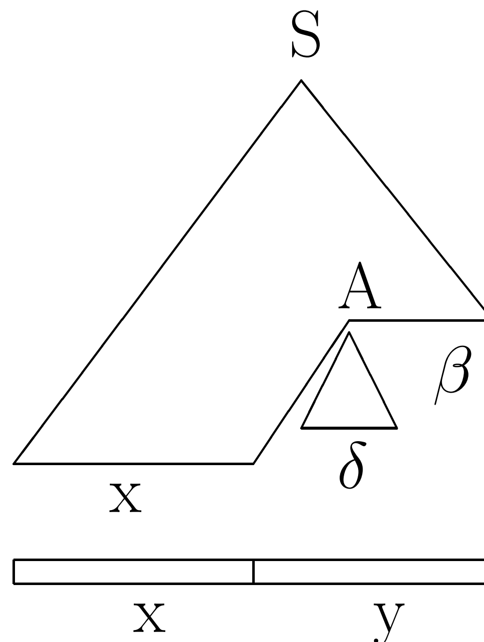
515: Programming Languages & Compilers I

- Sakai news group is open now
- Office hours have been posted
- First homework and project will be posted by Friday evening on our web site
- No class next Thursday; what are possible dates/times for a make-up class?

Review: Top-down parsers

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- some grammars are backtrack-free (*predictive*)
- *LL Parsing:* reads input from *left to right* and constructs *leftmost* derivation (forwards);
  LL Parsing is predictive.

$$S \Rightarrow_{lm}^{*} xA\beta \Rightarrow_{lm} x\delta\beta \Rightarrow_{lm}^{*} xy$$



- $x, y \in T^*; S, A \in NT; \delta, \beta \in (T \cup NT)^*; A \rightarrow \delta \in P$

Predictive Parsing — LL(1)

Basic idea:

> For any two productions $A ::= \alpha \mid \beta$, we would like a distinct way of choosing the correct production to expand.

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

That is
$x \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* x\gamma$ for some γ, and
$\epsilon \in \text{FIRST}(\alpha)$ *iff* $\alpha \Rightarrow^* \epsilon$.

For a non-terminal A, define **FOLLOW**(A) as the set of terminals that can appear immediately to the right of A in some sentential form.

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it

A terminal symbol has no FOLLOW set

Predictive Parsing — LL(1) (cont.)

Key Property:

Whenever two productions $A ::= \alpha$ and $A ::= \beta$ both appear in the grammar, we would like

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$, and
- if $\alpha \Rightarrow^* \epsilon$ then in addition to above condition: $FIRST(\beta) \cap FOLLOW(A) = \emptyset$
- Analogue case for $\beta \Rightarrow^* \epsilon$. Note: due to first condition, at most one of α or β can derive ϵ.

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

LL(1) grammars

Features

- input parsed from left to right
- leftmost derivation (forward)
- one token lookahead

Definition

A grammar G is $LL(1)$ if and only if for each set of productions $A ::= \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$

1. $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \cdots, \text{FIRST}(\alpha_n)$ are all pairwise disjoint, and
2. if $\alpha_i \Rightarrow^* \epsilon$, then in addition $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$, for all $1 \leq j \leq n, i \neq j$.

What rule to select for a given non-terminal and input token can be represented in a *parse table M*.

Algorithm for $LL(1)$ parse table construction must not result in multiple entries for any $M[A, a]$ or $M[A, eof]$ (Aho, Sethi, and Ullman, Algorithm 4.4).

$\Rightarrow$ Whether a grammar is $LL(1)$ or not is decidable.

Table-driven predictive parser — LL(1)

Input: a string w and a parsing table M for G

```
push eof
push Start Symbol
token ← next_token()
X ← top-of-stack
repeat
   if X is a terminal then
      if X = token then
         pop X
         token ← next_token()
      else error()
   else /* X is a non-terminal */
      if M[X,token] = X → Y1Y2···Yk then
         pop X
         push Yk, Yk−1, ···, Y1
      else error()
   X ← top-of-stack
until X = eof
if token ≠ eof then error()
```

Aho, Sethi, and Ullman, Algorithm 4.3

Example grammar and its table

expression grammar with precedence

$$
\begin{array}{lcl}
\langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\
\langle\text{expr}\rangle & ::= & \langle\text{term}\rangle\ \langle\text{expr}'\rangle \\
\langle\text{expr}'\rangle & ::= & \texttt{+}\ \langle\text{expr}\rangle \\
 & | & \texttt{-}\ \langle\text{expr}\rangle \\
 & | & \epsilon \\
\langle\text{term}\rangle & ::= & \langle\text{factor}\rangle\ \langle\text{term}'\rangle \\
\langle\text{term}'\rangle & ::= & \texttt{*}\ \langle\text{term}\rangle \\
 & | & \texttt{/}\ \langle\text{term}\rangle \\
 & | & \epsilon \\
\langle\text{factor}\rangle & ::= & \texttt{num} \\
 & | & \texttt{id}
\end{array}
$$

LL(1) parse table

	`id`	`num`	`+`	`-`	`*`	`/`	`eof`
$\langle\text{goal}\rangle$	$g \rightarrow e$	$g \rightarrow e$	–	–	–	–	–
$\langle\text{expr}\rangle$	$e \rightarrow te'$	$e \rightarrow te'$	–	–	–	–	–
$\langle\text{expr}'\rangle$	–	–	$e' \rightarrow \texttt{+}e$	$e' \rightarrow \texttt{-}e$	–	–	$e' \rightarrow \epsilon$
$\langle\text{term}\rangle$	$t \rightarrow ft'$	$t \rightarrow ft'$	–	–	–	–	–
$\langle\text{term}'\rangle$	–	–	$t' \rightarrow \epsilon$	$t' \rightarrow \epsilon$	$t' \rightarrow \texttt{*}t$	$t' \rightarrow \texttt{/}t$	$t' \rightarrow \epsilon$
$\langle\text{factor}\rangle$	$f \rightarrow \texttt{id}$	$f \rightarrow \texttt{num}$	–	–	–	–	–

Recursive descent parsing — LL(1)

Recursive descent is one of the simplest parsing techniques used in practical compilers:

- Each non–terminal has an associated parsing procedure that can recognize any sequence of tokens generated by that non–terminal.
- Within a parsing procedure, both non–terminals and terminals can be matched:
  - non–terminal A — call parsing procedure for A
  - token t — compare t with current input token; if match, consume input, otherwise ERROR
- Parsing procedures may contain code that performs some useful "computation" (syntax directed translation).

Recursive descent parser

For our example grammar

```
goal:
  token ← next_token();
  if (expr() = ERROR | token ≠ EOF) then
    return ERROR;
  else return OK;

expr:
  if (term() = ERROR) then
    return ERROR;
  else return expr_prime();

expr_prime:
  if (token = PLUS) then
    token ← next_token(); return expr();
  else if (token = MINUS) then
    token ← next_token(); return expr();
  else if (token = eof) then
    return OK;
  else return ERROR;
```

Recursive Descent Parsing (Cont.)

```
term:
  if (factor() = ERROR) then
    return ERROR;
  else return term_prime();

term_prime:
  if (token = MULT) then
    token ← next_token(); return term();
  else if (token = DIV) then
    token ← next_token(); return term();
  else if (token = eof) then
    return OK;
  if (token = PLUS) then
    return OK;
  if (token = MINUS) then
    return OK;
  else return ERROR;

factor:
  if (token = NUM) then
    token ← next_token(); return OK;
  else if (token = ID) then
    token ← next_token(); return OK;
  else return ERROR;
```

LL(1) grammars

Provable facts about *LL(1)* grammars:

- no left recursive grammar is *LL(1)*
- no ambiguous grammar is *LL(1)*
- *LL(1)* parsers operate in linear time
- an ϵ–free grammar where each alternative expansion for A begins with a distinct terminal is a *simple* *LL(1)* grammar

Not all grammars are *LL(1)*

- $S ::= aS \mid a$

  is not *LL(1)*

  $\text{FIRST}(aS) = \text{FIRST}(a) = \{a\}$

- $$\begin{array}{l} S ::= aS' \\ S' ::= aS' \mid \epsilon \end{array}$$

  accepts the same language and is *LL(1)*

LL grammars

LL(1) grammars

- may need to rewrite grammar
  (left recursion removal, left factoring)
- resulting grammar larger, less maintainable

LL(k) grammars

- k-token lookahead
- more powerful than *LL(1)* grammars
- example:
  $S ::= ac \mid abc$ is *LL(2)*

Not all grammars are *LL(k)*

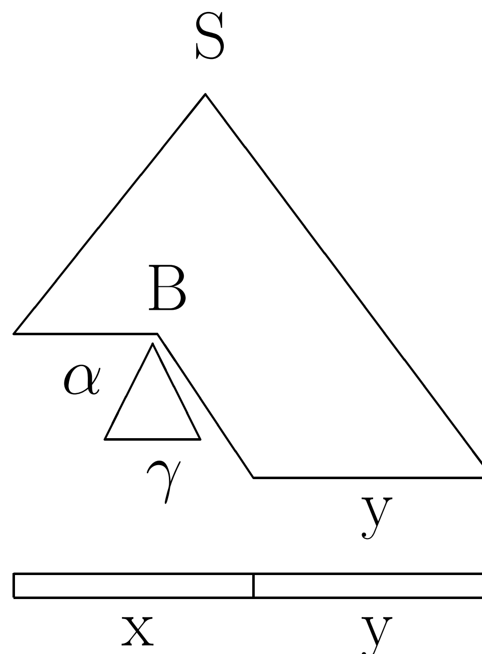
- example:
  Set of productions of form: $S ::= a^i b^j \quad$ for $i \geq j$
- problem - must choose production after k tokens of lookahead

Bottom-up parsers avoid some of these problems

Bottom-up parsers

- start at the leaves and fill in
- construct rightmost derivation in reverse
- find the next right-hand side of a production (*handle*) such that its replacement by left-hand side nonterminal will yield previous right-sentential form
- as input is consumed, change state to encode possibilities (*recognize valid prefixes*); if *handle* is found, `REDUCE`, otherwise `SHIFT` (or `ERROR`)

$$S \;\Rightarrow^*_{rm}\; \alpha B y \;\Rightarrow_{rm}\; \alpha\gamma y \;\Rightarrow^*_{rm}\; xy$$



- *LR parsing:* Reads input from *left to right* and constructs *rightmost* derivation in reverse

Example

Consider the context-free grammar (in BNF notation)

1	⟨goal⟩	::=	`a` ⟨A⟩ ⟨B⟩ `e`
2	⟨A⟩	::=	⟨A⟩ `b c`
3		|	`b`
4	⟨B⟩	::=	`d`

and the input string `abbcde`.

Prod'n.	Sentential Form	Handle†
—	a[b]bcde	(3,2)
3	a[⟨A⟩bc]de	(2,4)
2	a⟨A⟩[d]e	(4,3)
4	[a⟨A⟩⟨B⟩e]	(1,4)
1	⟨goal⟩	—

Why is (3,3) not a handle for a⟨A⟩[b]cde?

The trick appears to be scanning the input and finding valid right-sentential forms.

† *(rule, position of right end of handle in input string).*

Handles

We trying to find a substring α of the current right-sentential form where:

- α matches some production $A ::= \alpha$
- reducing α to A is one step in the reverse of a rightmost derivation.

We will call such a string a *handle*.

Formally,

- a *handle* of a right-sentential form γ is a production $A ::= \beta$ and a position in γ where β may be found. Convention: position sepcifies the right end of handle.
- If $(A ::= \beta, k)$ is a handle, then replacing the β in γ at position k with A produces the previous right-sentential form in a rightmost derivation of γ.

Handles

Provable fact:

> The substring to the right of a handle contains only terminal symbols.

Proof: Follows from the fact that all γ_i are right-sentential forms.

Corollary

> The right end of a handle is to the right of the previously reduced variable.

Shift-reduce parsing

One scheme to implement a handle-pruning, bottom-up parser is called a *shift-reduce* parser.

Shift-reduce parsers use a *stack* and an *input buffer*

1. initialize stack with $
2. Repeat until the top of the stack is the goal symbol and the input token is `eof`

   a) *find the handle*

   if we don't have a handle on top of the stack, *shift* an input symbol onto the stack

   b) *prune the handle*

   if we have a handle $(A ::= \beta, k)$ on top of the stack, *reduce*

   i) pop $\mid \beta \mid$ symbols off the stack

   ii) push A onto the stack

Example

Left-recursive expression grammar

(*original form*, before left recursion removal and factoring
→ our example LL(1) grammar on page 7.

1	⟨goal⟩	::=	⟨expr⟩
2	⟨expr⟩	::=	⟨expr⟩ `+` ⟨term⟩
3		|	⟨expr⟩ `-` ⟨term⟩
4		|	⟨term⟩
5	⟨term⟩	::=	⟨term⟩ `*` ⟨factor⟩
6		|	⟨term⟩ `/` ⟨factor⟩
7		|	⟨factor⟩
8	⟨factor⟩	::=	`num`
9		|	`id`

“`x - 2 * y`”

Stack	Input	Handle	Action
$	`id - num * id`	*none*	shift
$`id`	`- num * id`	9,1	reduce 9
$⟨factor⟩	`- num * id`	7,1	reduce 7
$⟨term⟩	`- num * id`	4,1	reduce 4
$⟨expr⟩	`- num * id`	*none*	shift
$⟨expr⟩ `-`	`num * id`	*none*	shift
$⟨expr⟩ `-` `num`	`* id`	8,3	reduce 8
$⟨expr⟩ `-` ⟨factor⟩	`* id`	7,3	reduce 7
$⟨expr⟩ `-` ⟨term⟩	`* id`	*none*	shift
$⟨expr⟩ `-` ⟨term⟩ `*`	`id`	*none*	shift
$⟨expr⟩ `-` ⟨term⟩ `*` `id`		9,5	reduce 9
$⟨expr⟩ `-` ⟨term⟩ `*` ⟨factor⟩		5,5	reduce 5
$⟨expr⟩ `-` ⟨term⟩		3,3	reduce 3
$⟨expr⟩		1,1	reduce 1
$⟨goal⟩		*none*	accept

1. *Shift until top of stack is the right end of a handle*
2. *Find the left end of the handle and reduce*

5 shifts + 9 reduces + 1 accept

Viable prefix

A *viable prefix* is

1. a prefix of a right-sentential form that does not continue past the right end of the handle of that sentential form†, or
2. a prefix of a right-sentential form that can appear on the stack of a shift-reduce parser.

It is always possible to add terminals onto the end of a viable prefix to obtain a right-sentential form.

As long as the prefix represented by the stack is viable, the parser has not seen a detectable error.

† If the grammar is unambiguous, there is a unique rightmost handle. *LR(k)* grammars are unambiguous.

Shift-reduce parsing

Grammars that are often used to construct shift-reduce parsers:

- operator grammars (will not discuss here
  $\rightarrow$ Aho, Sethi, Ullman p.203)
- LR(1) grammars
  - canonical LR(1) grammars
  - simple LR(1) grammars (*SLR(1)*)
  - lookahead LR(1) grammars (*LALR(1)*)

Grammars use different methods or levels of "context" information to detect handle.

LR(1), *SLR(1)* and *LALR(1)*) grammars use finite automata (*NFA*s or *DFA*s) to recognize viable prefixes and store "context" information.

LR(k) grammars

Informally, we say that a grammar G is LR(k) if,

given a rightmost derivation

$$S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \cdots \Rightarrow_{rm} \gamma_n = w ,$$

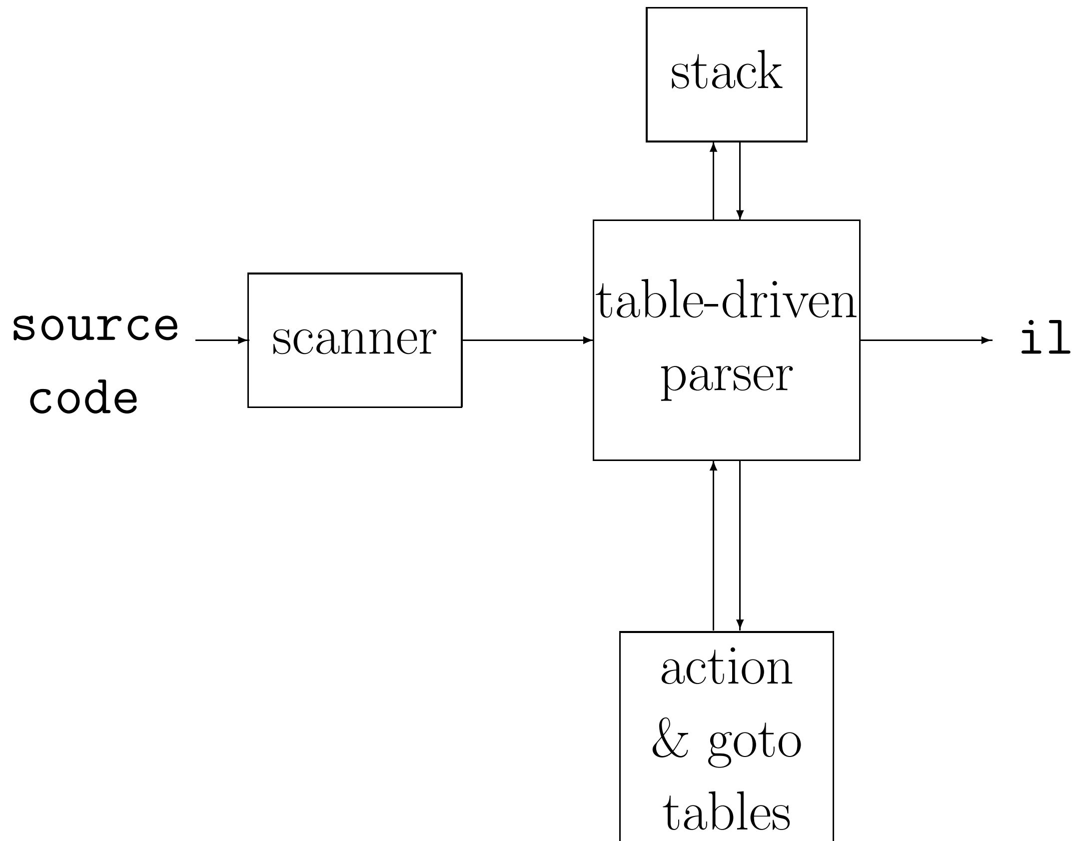
we can, for each right-sentential form in the derivation,

1. *isolate the handle of each right-sentential form,* and
2. *determine the production by which to reduce*

by scanning γ_i from left to right, going at most k symbols beyond the right end of the handle of γ_i.

Table-driven *LR* parsing

A table-driven LR(k) parser looks like



Stack two items per state: *state* and *symbol*

Why study LR(1) grammars?

- All context-free, deterministic languages have an LR(1) grammar. Therefore LR grammars describe a proper superset of the languages recognized by LL (predictive) parsers.
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser
- Efficient shift-reduce parsers can be implemented for LR(1) grammars — time proportional to $tokens + reductions$
- Easy to build since table construction can be automated
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- Everyone's favorite parser (*EFP*) — tools widely available (example: `yacc`).

$LR(1)$ parsing

The skeleton parser:

```
token = next_token()
repeat forever
  s = state on top of stack
  if action[s,token] = "shift s_i"' then
    push token
    push state s_i
    token = next_token()
  else if action[s,token] =
      "reduce A ::= β" then
    pop 2 * | β | symbols
    s = state on top of stack
    push A
    push goto[s,A]
  else if action[s, token] = "accept" then
    return
  else error()
```

This takes k shifts, l reduces, and 1 accept, where k is the length of the input string and l is the length of the reverse rightmost derivation.

Note: See Figure 4.36, Aho, Lam, Sethi, and Ullman

LR(0) parsing: our example

Stack (without states)	Input	Handle	Action
$	`id - num * id`	*none*	shift
$`id`	`- num * id`	9,1	reduce 9
$⟨factor⟩	`- num * id`	7,1	reduce 7
$⟨term⟩	`- num * id`	4,1	reduce 4
$⟨expr⟩	`- num * id`	*none*	shift
$⟨expr⟩ `-`	`num * id`	*none*	shift
$⟨expr⟩ `- num`	`* id`	8,3	reduce 8
$⟨expr⟩ `-` ⟨factor⟩	`* id`	7,3	reduce 7
$⟨**expr**⟩ `-` ⟨**term**⟩	`* id`	*none*	shift
$⟨expr⟩ `-` ⟨term⟩ `*`	`id`	*none*	shift
$⟨expr⟩ `-` ⟨term⟩ `* id`		9,5	reduce 9
$⟨expr⟩ `-` ⟨term⟩ `*` ⟨factor⟩		5,5	reduce 5
$⟨**expr**⟩ `-` ⟨**term**⟩		3,3	reduce 3
$⟨expr⟩		1,1	reduce 1
$⟨goal⟩		*none*	accept

The corresponding grammar and language is not LR(0).

Theorem: A language L has an LR(0) grammar iff

- L is deterministic
- no proper prefix of a word in L is in L (*prefix property*)

Next class: Syntax Directed Translation and Project

For next time, please read: Scott: Chapters 4, 5.1-5.4;

First homework will be out soon. Please see our course website.