CMSC 430
Introduction to Compilers
Spring 2017

Lexing and Parsing
Overview

• Compilers are roughly divided into two parts
  ▪ Front-end — deals with surface syntax of the language
  ▪ Back-end — analysis and code generation of the output of the front-end

• Lexing and Parsing translate source code into form more amenable for analysis and code generation

• Front-end also may include certain kinds of semantic analysis, such as symbol table construction, type checking, type inference, etc.
Lexing vs. Parsing

- Language grammars usually split into two levels
  - Tokens — the “words” that make up “parts of speech”
    - Ex: Identifier \[a-zA-Z_]+\[
    - Ex: Number \[0-9]+\[
  - Programs, types, statements, expressions, declarations, definitions, etc — the “phrases” of the language
    - Ex: if (expr) expr;
    - Ex: def id(id, ..., id) expr end

- Tokens are identified by the lexer
  - Regular expressions

- Everything else is done by the parser
  - Uses grammar in which tokens are primitives
  - Implementations can look inside tokens where needed
Lexing vs. Parsing (cont’d)

• Lexing and parsing often produce abstract syntax tree as a result
  ▪ For efficiency, some compilers go further, and directly generate intermediate representations

• Why separate lexing and parsing from the rest of the compiler?
• Why separate lexing and parsing from each other?
Parsing theory

• Goal of parsing: Discovering a parse tree (or derivation) from a sentence, or deciding there is no such parse tree

• There’s an alphabet soup of parsers
  ▪ Cocke-Younger-Kasami (CYK) algorithm; Earley’s Parser
    - Can parse any context-free grammar (but inefficient)
  ▪ LL(k)
    - top-down, parses input left-to-right (first L), produces a leftmost derivation (second L), k characters of lookahead
  ▪ LR(k)
    - bottom-up, parses input left-to-right (L), produces a rightmost derivation (R), k characters of lookahead

• We will study only some of this theory
  ▪ But we’ll start more concretely
Parsing practice

• Yacc and lex — most common ways to write parsers
  ▪ yacc = “yet another compiler compiler” (but it makes parsers)
  ▪ lex = lexical analyzer (makes lexers/tokenizers)

• These are available for most languages
  ▪ bison/flex — GNU versions for C/C++
  ▪ ocamlyacc/ocamllex — what we’ll use in this class
Example: Arithmetic expressions

- High-level grammar:
  - $$E \rightarrow E + E \mid n \mid (E)$$
- What should the tokens be?
  - Typically they are the terminals in the grammar
    - $$\{+, (, ), n\}$$
    - Notice that $$n$$ itself represents a set of values
    - Lexers use regular expressions to define tokens
  - But what will a typical input actually look like?
    
    ```
    1 + 2 + \n ( 3 + 4 2 )
    ```
    - We probably want to allow for whitespace
      - Notice not included in high-level grammar: lexer can discard it
    - Also need to know when we reach the end of the file
      - The parser needs to know when to stop
Lexing with ocamllex (.mll)

```ocaml
(* Slightly simplified format *)
{ header }
rule entrypoint = parse
    regexp_1 { action_1 }
    | ...
    | regexp_n { action_n }
and ...
{ trailer }
```

- Compiled to .ml output file
  - **header** and **trailer** are inlined into output file as-is
  - **regexps** are combined to form one (big!) finite automaton that recognizes the union of the regular expressions
    - Finds *longest* possible match in the case of multiple matches
    - Generated regexp matching function is called **entrypoint**
Lexing with ocamllex (.mll)

• When match occurs, generated **entrypoint** function returns value in corresponding action
  - If we are lexing for **ocamlyacc**, then we’ll return tokens that are defined in the **ocamlyacc** input grammar
Example

```ocaml
{
    open Ex1_parser
    exception Eof
}
rule token = parse
    | [' ' '	' '']       { token lexbuf } (* skip blanks *)
    | ['\n']              { EOL }
    | ['0'-'9']+ as lxm    { INT(int_of_string lxm) }
    | '+'                 { PLUS }
    | '('                 { LPAREN }
    | ')'                 { RPAREN }
    | eof                 { raise Eof }

(* token definition from Ex1_parser *)
type token =
    | INT of (int)
    | EOL
    | PLUS
    | LPAREN
    | RPAREN
```
Generated code

You don’t need to understand the generated code
  - But you should understand it’s not magic

Uses **Lexing** module from OCaml standard lib

Notice that **token** rule was compiled to **token** fn
  - Mysterious **lexbuf** from before is the argument to **token**
  - Type can be examined in **Lexing** module ocamldoc
Lexer limitations

- Automata limited to 32767 states
  - Can be a problem for languages with lots of keywords

```plaintext
rule token = parse
  "keyword_1"   { ... }
| "keyword_2"   { ... }
| ...
| "keyword_n"  { ... }
| ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '_'] * as id
  { IDENT id}
```

- Solution?
Parsing

• Now we can build a parser that works with lexemes (tokens) from `token.mll`
  - Recall from 330 that parsers work by consuming one character at a time off input while building up parse tree
  - Now the input stream will be tokens, rather than chars

```
  1   +   2   +   \n   (   3   +   4   2   )   eof

  INT(1)   PLUS   INT(2)   PLUS   LPAREN   INT(3)   PLUS   INT(42)   RPAREN   eof
```

- Notice parser doesn’t need to worry about whitespace, deciding what’s an `INT`, etc
Suitability of Grammar

• Problem: our grammar is ambiguous
  - $E \rightarrow E + E \mid n \mid (E)$
  - Exercise: find an input that shows ambiguity

• There are parsing technologies that can work with ambiguous grammars
  - But they’ll provide multiple parses for ambiguous strings, which is probably not what we want

• Solution: remove ambiguity
  - One way to do this from 330:
    - $E \rightarrow T \mid E + T$
    - $T \rightarrow n \mid (E)$
Parsing with ocamlyacc (.mly)

• Compiled to .ml and .mli files
  - .mli file defines token type and entry point main for parsing
    - Notice first arg to main is a fn from a lexbuf to a token, i.e., the function generated from a .mll file!
Parsing with ocamlyacc (.mly)

- .ml file uses Parsing library to do most of the work
  - header and trailer copied direct to output
  - declarations lists tokens and some other stuff
  - rules are the productions of the grammar
    - Compiled to yytables; this is a table-driven parser Also include actions that are executed as parser executes
    - We’ll see an example next
Actions

• In practice, we don’t just want to check whether an input parses; we also want to do something with the result
  ▪ E.g., we might build an AST to be used later in the compiler
• Thus, each production in ocamlyacc is associated with an action that produces a result we want
• Each rule has the format
  ▪ lhs: rhs \{act\}
  ▪ When parser uses a production $\text{lhs} \rightarrow \text{rhs}$ in finding the parse tree, it runs the code in act
  ▪ The code in act can refer to results computed by actions of other non-terminals in rhs, or token values from terminals in rhs
Example

```pascal
%token <int> INT
%token EOL PLUS LPAREN RPAREN
%start main /* the entry point */
%type <int> main
%
main:
| expr EOL              { $1 }        (* 1 *)
expr:
| term                  { $1 }        (* 2 *)
| expr PLUS term        { $1 + $3 }   (* 3 *)
term:
| INT                   { $1 }        (* 4 *)
| LPAREN expr RPAREN    { $2 }        (* 5 *)
```

- Several kinds of declarations:
  - `%token` — define a token or tokens used by lexer
  - `%start` — define start symbol of the grammar
  - `%type` — specify type of value returned by actions
## Actions, in action

<table>
<thead>
<tr>
<th>INT(1)</th>
<th>PLUS</th>
<th>INT(2)</th>
<th>PLUS</th>
<th>LPAREN</th>
<th>INT(3)</th>
<th>PLUS</th>
<th>INT(42)</th>
<th>RPAREN</th>
<th>eof</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
.1+2+(3+42)$

term[1].+2+(3+42)$

expr[1].+2+(3+42)$


expr[3].+(3+42)$


expr[3]+(expr[45].)$


expr[48].$

main[48]
```

**main:**
- `| expr EOL  { $1 }`
- `| expr PLUS term  { $1 + $3 }`
- `| term  { $1 }`
- `| LPAREN expr RPAREN  { $2 }`

- The “.” indicates where we are in the parse
- We’ve skipped several intermediate steps here, to focus only on actions
- (Details next)
Actions, in action

```
main:  
  | expr EOL          { $1 }  
expr:  
  | term              { $1 }  
  | expr PLUS term    { $1 + $3 }  
term:  
  | INT               { $1 }  
  | LPAREN expr RPAREN { $2 }  

1 INT(1) PLUS INT(2) PLUS LPAREN INT(3) PLUS INT(42) RPAREN eof
```

```
main[48]
     +
expr[48]
     +
  +
  +
2 term[1]
  +
3 expr[3]
  +
42 term[42]
  (expr[45]
```
Invoking lexer/parser

```ocaml
try
  let lexbuf = Lexing.from_channel stdin in
  while true do
    let result = Ex1_parser.main Ex1_lexer.token lexbuf in
    print_int result; print_newline(); flush stdout
  done
with Ex1_lexer.Eof ->
  exit 0
```

- Tip: can also use `Lexing.from_string` and `Lexing.from_function`
Terminology review

• Derivation
  - A sequence of steps using the productions to go from the start symbol to a string

• Rightmost (leftmost) derivation
  - A derivation in which the rightmost (leftmost) nonterminal is rewritten at each step

• Sentential form
  - A sequence of terminals and non-terminals derived from the start-symbol of the grammar with 0 or more reductions
  - I.e., some intermediate step on the way from the start symbol to a string in the language of the grammar

• Right- (left-)sentential form
  - A sentential form from a rightmost (leftmost) derivation

• FIRST(\(\alpha\))
  - Set of initial symbols of strings derived from \(\alpha\)
Bottom-up parsing

• ocamlyacc builds a bottom-up parser
  ▪ Builds derivation from input back to start symbol
    \[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{input} \]

• To reduce \( \gamma_i \) to \( \gamma_{i-1} \)
  ▪ Find production \( A \rightarrow \beta \) where \( \beta \) is in \( \gamma_i \), and replace \( \beta \) with \( A \)

• In terms of parse tree, working from leaves to root
  ▪ Nodes with no parent in a partial tree form its upper fringe
  ▪ Since each replacement of \( \beta \) with \( A \) shrinks upper fringe, we call it a reduction.

• Note: need not actually build parse tree
  ▪ \(|\text{parse tree nodes}| = |\text{input}| + |\text{reductions}|\)
Bottom-up parsing, illustrated

LR(1) parsing
• Scan input left-to-right
• Rightmost derivtaion
• 1 token lookahead

[Diagram]

Rule: $B \rightarrow \gamma$

S $\Rightarrow^* \alpha \ B \ y \ ⇒ \ \alpha \ \gamma \ y \ ⇒^* \ x \ y$

Upper fringe: solid
Yet to be parsed: dashed
**Bottom-up parsing, illustrated**

LR(1) parsing
- Scan input left-to-right
- Rightmost derivation
- 1 token lookahead

\[ S \Rightarrow^* \alpha B y \Rightarrow \alpha \gamma y \Rightarrow^* x y \]

**Upper fringe: solid**
**Yet to be parsed: dashed**
Finding reductions

• Consider the following grammar

  1. S → a A B e
  2. A → A b c
  3. | b
  4. B → d

  Input: abbcde

<table>
<thead>
<tr>
<th>Sentential Form</th>
<th>Production</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbcde</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>aAbcde</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>aAde</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>aABe</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>S</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

• How do we find the next reduction?
  • How do we do this efficiently?
Handles

• Goal: Find substring $\beta$ of tree’s frontier that matches some production $A \rightarrow \beta$
  ▪ (And that occurs in the rightmost derivation)
  ▪ Informally, we call this substring $\beta$ a *handle*

• Formally,
  ▪ A *handle* of a right-sentential form $\gamma$ is a pair $(A \rightarrow \beta,k)$ where
    - $A \rightarrow \beta$ is a production and $k$ is the position in $\gamma$ of $\beta$’s rightmost symbol.
    - If $(A \rightarrow \beta,k)$ is a handle, then replacing $\beta$ at $k$ with $A$ produces the right
      sentential form from which $\gamma$ is derived in the rightmost derivation.
  ▪ Because $\gamma$ is a right-sentential form, the substring to the right of a handle contains only terminal symbols
    - $\Rightarrow$ the parser doesn’t need to scan past the handle (only lookahead)
Example

- Grammar

1. \( S \rightarrow E \)
2. \( E \rightarrow E + T \)
3. \( | E - T \)
4. \( | T \)
5. \( T \rightarrow T * F \)
6. \( | T / F \)
7. \( | F \)
8. \( F \rightarrow n \)
9. \( | id \)
10. \( | (E) \)

<table>
<thead>
<tr>
<th>Production</th>
<th>Sentential Form</th>
<th>Handle (prod,k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>E</td>
<td>1,1</td>
</tr>
<tr>
<td>3</td>
<td>E-T</td>
<td>3,3</td>
</tr>
<tr>
<td>5</td>
<td>E-T*F</td>
<td>5,5</td>
</tr>
<tr>
<td>9</td>
<td>E-T*id</td>
<td>9,5</td>
</tr>
<tr>
<td>7</td>
<td>E-F*id</td>
<td>7,3</td>
</tr>
<tr>
<td>8</td>
<td>E-n*id</td>
<td>8,3</td>
</tr>
<tr>
<td>4</td>
<td>T-n*id</td>
<td>4,1</td>
</tr>
<tr>
<td>7</td>
<td>F-n*id</td>
<td>7,1</td>
</tr>
<tr>
<td>9</td>
<td>id-n*id</td>
<td>9,1</td>
</tr>
</tbody>
</table>

Handles for rightmost derivation of \( id-n*id \)
Finding reductions

- Theorem: If $G$ is unambiguous, then every right-sentential form has a unique handle
  - If we can find those handles, we can build a derivation!

- Sketch of Proof:
  - $G$ is unambiguous $\Rightarrow$ rightmost derivation is unique
  - $\Rightarrow$ a unique production $A \rightarrow \beta$ applied to derive $\gamma_i$ from $\gamma_{i-1}$
  - and a unique position $k$ at which $A \rightarrow \beta$ is applied
  - $\Rightarrow$ a unique handle $(A \rightarrow \beta, k)$

- This all follows from the definitions
Bottom-up handle pruning

- **Handle pruning**: discovering handle and reducing it
  - Handle pruning forms the basis for bottom-up parsing
- So, to construct a rightmost derivation
  \[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{input} \]
- Apply the following simple algorithm

  ```plaintext
  for i ← n to 1 by −1
  Find handle \( (A_i \rightarrow \beta_i, k_i) \) in \( \gamma_i \)
  Replace \( \beta_i \) with \( A_i \) to generate \( \gamma_{i-1} \)
  ```
  - This takes \( 2n \) steps
Shift-reduce parsing algorithm

- Maintain a stack of terminals and non-terminals matched so far
  - Rightmost terminal/non-terminal on top of stack
  - Since we’re building rightmost derivation, will look at top elements of stack for reductions

```plaintext
push INVALID
token ← next_token()
repeat until (top of stack = Goal and token = EOF)
  if the top of the stack is a handle A→β
    then // reduce β to A
      pop |β| symbols off the stack
      push A onto the stack
  else if (token ≠ EOF)
    then // shift
      push token
      token ← next_token()
  else // need to shift, but out of input
    report an error
```

Potential errors

- Can’t find handle
- Reach end of file
Example

- Grammar
  1. $S \rightarrow E$
  2. $E \rightarrow E + T$
  3. $| E - T$
  4. $| T$
  5. $T \rightarrow T * F$
  6. $| T / F$
  7. $| F$
  8. $F \rightarrow n$
  9. $| id$
  10. $| (E)$

Shift/reduce parse of id-n*id

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle (prod,k)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>id-n*id</td>
<td>none</td>
<td>shift</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>-n*id</td>
<td>9,1</td>
<td>reduce 9</td>
</tr>
<tr>
<td>F</td>
<td>-n*id</td>
<td>7,1</td>
<td>reduce 7</td>
</tr>
<tr>
<td>T</td>
<td>-n*id</td>
<td>4,1</td>
<td>reduce 4</td>
</tr>
<tr>
<td>E</td>
<td>-n*id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>E-n</td>
<td>*id</td>
<td>8,3</td>
<td>reduce 8</td>
</tr>
<tr>
<td>E-F</td>
<td>*id</td>
<td>7,3</td>
<td>reduce 7</td>
</tr>
<tr>
<td>E-T</td>
<td>*id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>E-T*</td>
<td>id</td>
<td>none</td>
<td>shift</td>
</tr>
<tr>
<td>E-T*id</td>
<td>9,5</td>
<td>reduce 9</td>
<td></td>
</tr>
<tr>
<td>E-T*F</td>
<td>5,5</td>
<td>reduce 5</td>
<td></td>
</tr>
<tr>
<td>E-T</td>
<td>3,3</td>
<td>reduce 3</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>1,1</td>
<td>reduce 1</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>none</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce
Parse tree for example
Algorithm actions

• Shift-reduce parsers have just four actions
  ▪ **Shift** — next word is shifted onto the stack
  ▪ **Reduce** — right end of handle is at top of stack
    - Locate left end of handle within the stack
    - Pop handle off stack and push appropriate lhs
  ▪ **Accept** — stop parsing and report success
  ▪ **Error** — call an error reporting/recovery routine

• Cost of operations
  ▪ **Accept** is constant time
  ▪ **Shift** is just a push and a call to the scanner
  ▪ **Reduce** takes $|\text{rhs}|$ pops and 1 push
    - If handle-finding requires state, put it in the stack $\Rightarrow 2x$ work
  ▪ **Error** depends on error recovery mechanism
Finding handles

• To be a handle, a substring of sentential form \( \gamma \) must:
  - Match the right hand side \( \beta \) of some rule \( A \rightarrow \beta \)
  - There must be some rightmost derivation from the start symbol that produces \( \gamma \) with \( A \rightarrow \beta \) as the last production applied
  - \( \Rightarrow \) Looking for rhs’s that match strings is not good enough

• How can we know when we have found a handle?
  - LR(1) parsers use DFA that runs over stack and finds them
    - One token look-ahead determines next action (shift or reduce) in each state of the DFA.
  - A grammar is LR(1) if we can build an LR(1) parser for it
• LR(0) parsers: no look-ahead
• Can use a set of tables to describe LR(1) parser

- ocamlyacc automates the process of building the tables
  - Standard library Parser module interprets the tables
- LR parsing invented in 1965 by Donald Knuth
- LALR parsing invented in 1969 by Frank DeRemer
LR(1) parsing algorithm

- Two tables
  - ACTION: reduce/shift/accept
  - GOTO: state to be in after reduce
- Cost
  - |input| shifts
  - |derivation| reductions
  - One accept
- Detects errors by failure to shift, reduce, or accept

```python
code =
stack.push(INVALID); stack.push(s_0);
not_found = true;
token = scanner.next_token();
do while (not_found) {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) {
        stack.popnum(2*|β|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);
        stack.push(GOTO[s,A]);
    } else if ( ACTION[s,token] == "shift s_i" ) {
        stack.push(token); stack.push(s_i);
        token ← scanner.next_token();
    } else if ( ACTION[s,token] == "accept" && token == EOF )
        not_found = false;
else report a syntax error and recover;
}
report success;
```
Example parser table

- `ocamlyacc -v ex1_parser.mly` — produce `.output` file with parser table

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
<th>goto</th>
<th>productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>(special)</td>
</tr>
<tr>
<td>1</td>
<td>s3</td>
<td>s4</td>
<td>acc 6 7  <code>entry → . main</code></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>(special)</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
<td></td>
<td>term → INT .</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>s4</td>
<td>8 7  <code>term → ( . expr )</code></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>(special)</td>
</tr>
<tr>
<td>6</td>
<td>s9</td>
<td>s10</td>
<td>main → expr . EOL</td>
</tr>
<tr>
<td>7</td>
<td>r2</td>
<td></td>
<td>expr → term .</td>
</tr>
<tr>
<td>8</td>
<td>s10</td>
<td>s11</td>
<td>expr → expr . + term</td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td></td>
<td>main → expr EOL .</td>
</tr>
<tr>
<td>10</td>
<td>s3</td>
<td>s4</td>
<td>12  <code>expr → expr + . term</code></td>
</tr>
<tr>
<td>11</td>
<td>r5</td>
<td></td>
<td>term → ( expr ) .</td>
</tr>
<tr>
<td>12</td>
<td>r3</td>
<td></td>
<td><code>expr → expr + term .</code></td>
</tr>
</tbody>
</table>

NB: Numbers in shift refer to state numbers
Numbers in reduction refer to production numbers
## Example parse (N+N+N)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N+N+N</td>
<td>s3</td>
</tr>
<tr>
<td>1,N,3</td>
<td>+N+N</td>
<td>r4</td>
</tr>
<tr>
<td>1,term,7</td>
<td>+N+N</td>
<td>r2</td>
</tr>
<tr>
<td>1,expr,6</td>
<td>+N+N</td>
<td>s10</td>
</tr>
<tr>
<td>1,expr,6,+10</td>
<td>N+N</td>
<td>s3</td>
</tr>
<tr>
<td>1,expr,6,+10,N,3</td>
<td>+N</td>
<td>r4</td>
</tr>
<tr>
<td>1,expr,6,+10,term,12</td>
<td>+N</td>
<td>r3</td>
</tr>
<tr>
<td>1,expr,6</td>
<td>+N</td>
<td>s10</td>
</tr>
<tr>
<td>1,expr,6,+10</td>
<td>N</td>
<td>s3</td>
</tr>
<tr>
<td>1,expr,6,+10,N,3</td>
<td></td>
<td>r4</td>
</tr>
<tr>
<td>1,expr,6,+10,term,12</td>
<td></td>
<td>r3</td>
</tr>
<tr>
<td>1,expr,6</td>
<td></td>
<td>s9</td>
</tr>
<tr>
<td>1,expr,6,EOL,9</td>
<td></td>
<td>r1</td>
</tr>
<tr>
<td>accept</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example parser table (cont’d)

• Notes
  - Notice derivation is built up (bottom to top)
  - Table only contains kernel of each state
    - Apply closure operation to see all the productions in the state

• LR(1) parsing requires start symbol not on any rhs
  - Thus, ocamlyacc actually adds another production
    - `%entry% → \001 main`
    - (so the `acc` in the previous table is a slight fib)

• Values returned from actions stored on the stack
  - Reduce triggers computation of action result
Why does this work?

• Stack = upper fringe
  ▪ So all possible handles on top of stack
  ▪ Shift inputs until top elements of stack form a handle

• Build a handle-recognizing DFA
  ▪ Language of handles is regular
  ▪ ACTION and GOTO tables encode the DFA
    - Shift = DFA transition
    - Reduce = DFA accept
      - New state = GOTO[state at top of stack (after pop), lhs]

• If we can build these tables, grammar is LR(1)
LR(k) items

- An LR(k) item is a pair \([P, \delta]\), where
  - \(P\) is a production \(A \rightarrow \beta\) with a \(\cdot\) at some position in the rhs
  - \(\delta\) is a lookahead string of length \(\leq k\) (words or $\$
  - The \(\cdot\) in an item indicates the position of the top of the stack
- LR(1):
  - \([A \rightarrow \cdot \beta \gamma, a]\) — input so far consistent with using \(A \rightarrow \beta \gamma\) immediately after symbol on top of stack
  - \([A \rightarrow \beta \cdot \gamma, a]\) — input so far consistent with using \(A \rightarrow \beta \gamma\) at this point in the parse, and parser has already recognized \(\beta\)
  - \([A \rightarrow \beta \gamma \cdot, a]\) — parser has seen \(\beta \gamma\), and lookahead of a consistent with reducing to \(A\)
- LR(1) items represent valid configurations of an LR(1) parser; DFA states are sets of LR(1) items
LR(k) items, cont’d

• Ex: \( A \rightarrow BCD \) with lookahead a can yield 4 items
  - \([A \rightarrow \cdot BCD, a], [A \rightarrow B \cdot CD, a], [A \rightarrow BC \cdot D, a], [A \rightarrow BCD \cdot, a]\)
  - Notice: set of LR(1) items for a grammar is finite

• Carry lookaheads along to choose correct reduction
  - Lookahead has no direct use in \([A \rightarrow \beta \cdot \gamma, a]\)
  - In \([A \rightarrow \beta \cdot, a]\), a lookahead of \(a \Rightarrow \) reduction by \(A \rightarrow \beta\)
  - For \(\{ [A \rightarrow \beta \cdot, a], [B \rightarrow \gamma \cdot \delta, b] \}\)
    - Lookahead of \(a \Rightarrow \) reduce to \(A\)
    - \(\text{FIRST}(\delta) \Rightarrow\) shift
    - (else error)
LR(1) table construction

- States of LR(1) parser contain sets of LR(1) items
  - Initial state $s_0$
    - Assume $S'$ is the start symbol of grammar, does not appear in rhs
      - (Extend grammar if necessary to ensure this)
    - $s_0 = \text{closure}([S' \rightarrow \bullet S, \$]) \quad (\$ = \text{EOF})$
  - For each $s_k$ and each terminal/non-terminal $X$, compute new state $\text{goto}(s_k, X)$
    - Use $\text{closure}()$ to "fill out" kernel of new state
    - If the new state is not already in the collection, add it
    - Record all the transitions created by $\text{goto}()$
      - These become ACTION and GOTO tables
      - i.e., the handle-finding DFA
  - This process eventually reaches a fixpoint
Closure()

• \([A\to \beta \cdot B \delta, a]\) implies \([B\to \cdot \gamma, x]\) for each production with \(B\) on lhs and each \(x \in \text{FIRST}(\delta a)\)
  - (If you’re about to see a \(B\), you may also see a \(\gamma\))

```
Closure( s )
while ( s is still changing )
  \forall \text{ items } [A \to \beta \cdot B \delta, a] \in s // item with \cdot to left of nonterminal B
  \forall \text{ productions } B \to \gamma \in P // all productions for B
  \forall b \in \text{FIRST}(\delta a) // tokens appearing after B
  \text{ if } [B \to \cdot \gamma, b] \not\in s // form LR(1) item w/ new lookahead
  \text{ then add } [B \to \cdot \gamma, b] \text{ to } s // add item to } s \text{ if new}
```

• Classic fixed-point method
• Halts because \(s \subset \text{ITEMS}\) (worklist version is faster)
  • Closure “fills out” a state
Example — closure with LR(0)

S → E
E → T+E
| T
T → id

[S → • E]
[E → • T+E]
[E → • T]
[T → • id]

[E → T+ • E]
[E → • T+E]
[E → • T]
[T → • id]
Example — closure with LR(1)

S → E
E → T+E
  | T
T → id

 kernel item
 [S → • E, $]
 [E → • T+E, $]
 [E → • T, $]
 [T → • id, +]
 [T → • id, $]

 derived item
 [E → T+ • E, $]
 [E → • T+ E, $]
 [E → • T, $]
 [T → • id, +]
 [T → • id, $]
**Goto**

- **Goto**(*s, x*) computes the state that the parser would reach if it recognized an *x* while in state *s*
  
  - *Goto*( { [A→β•Xδ,a] }, X ) produces [A→βX•δ,a]
  
  - Should also includes closure( [A→βX•δ,a] )

```plaintext
Goto( s, X )
new ← Ø
∀ items [A→β•Xδ,a] ∈ s  // for each item with • to left of X
  new ← new ∪ [A→βX•δ,a]  // add item with • to right of X
return closure(new)  // remember to compute closure!
```

- Not a fixed-point method!
- Straightforward computation
- Uses closure ( )
  - Goto() moves forward
Example — goto with LR(0)

S → E
E → T+E
| T
T → id

[S → E •]
[E → T • +E]
[E → T •]
[T → id •]

[kernel item]
[derived item]
Example — goto with LR(1)

S → E
E → T+E
  |  T
T → id

[S → • E, $]
[E → • T+E, $]
[E → • T, $]
[T → • id, +]
[T → • id, $]

[S → E •, $]
[E → T • +E, $]
[E → T •, $]

[T → id •, +]
[T → id •, $]
Building parser states

\[ cc_0 \leftarrow \text{closure} \left( [S' \rightarrow \cdot S, \$] \right) \]
\[ \text{CC} \leftarrow \{ cc_0 \} \]

while (new sets are still being added to \text{CC})
for each unmarked set \( cc_j \in \text{CC} \)
    mark \( cc_j \) as processed
    for each \( x \) following a \( \cdot \) in an item in \( cc_j \)
        temp \leftarrow \text{goto}(cc_j, x)
        if temp \( \notin \) \text{CC}
            then \text{CC} \leftarrow \text{CC} \cup \{ temp \}
        record transitions from \( cc_j \) to temp on \( x \)

- \text{CC} = \text{canonical collection (of LR(k) items)}
- Fixpoint computation (worklist version)
- Loop adds to \text{CC}
  - \( \text{CC} \subseteq 2^{\text{ITEMS}}, \text{so CC is finite} \)
Example LR(0) states

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow T+E \\
| & \quad T \\
T & \rightarrow \text{id}
\end{align*}
\]

\[
\begin{align*}
[S & \rightarrow \bullet E] \\
[E & \rightarrow \bullet T+E] \\
[E & \rightarrow \bullet T] \\
[T & \rightarrow \bullet \text{id}]
\end{align*}
\]

\[
\begin{align*}
[E & \rightarrow T \bullet +E] \\
[E & \rightarrow T \bullet]
\end{align*}
\]

\[
\begin{align*}
[E & \rightarrow T + \bullet E] \\
[E & \rightarrow \bullet T+E] \\
[E & \rightarrow \bullet T] \\
[T & \rightarrow \bullet \text{id}]
\end{align*}
\]

\[
\begin{align*}
[E & \rightarrow T + E \bullet]
\end{align*}
\]

\[
\begin{align*}
[S & \rightarrow E \bullet] \\
[T & \rightarrow \text{id} \bullet]
\end{align*}
\]
Example LR(1) states

S → E
E → T+E
| T
T → id

[S → • E, $]
[E → • T+E, $]
[E → • T, $]
[T → • id, +]
[T → • id, $]

T

[E → • T+E, $]
[E → • T, $]
[T → • id, +]
[T → • id, $]

id

[E → T •, $]

id

[T → id •, +]
[T → id •, $]

E

[S → E •, $]

[E → T + E •, $]

E

[E → T + • E, $]
Building ACTION and GOTO tables

∀ set \( s_x \in S \)
∀ item \( i \in s_x \)
  if \( i \) is \([A\rightarrow\beta \cdot \alpha, a]\) and \( \text{goto}(s_x, a) = s_k, a \in \text{terminals} \) // • to left of terminal a
    then \( \text{ACTION}[x, a] \leftarrow \text{"shift } k\text{"} \) // ⇒ shift if lookahead = a
  else if \( i \) is \([S'\rightarrow S \cdot, $]\)
    then \( \text{ACTION}[x, $] \leftarrow \text{"accept"} \) // ⇒ accept if lookahead = $
  else if \( i \) is \([A\rightarrow\beta \cdot, a]\)
    then \( \text{ACTION}[x, a] \leftarrow \text{"reduce } A\rightarrow\beta\" \) // → production done
∀ \( n \in \text{nonterminals} \)
  if \( \text{goto}(s_x, n) = s_k \)
    then \( \text{GOTO}[x, n] \leftarrow k \) // store transitions for nonterminals

• Many items generate no table entry
  e.g., \([A\rightarrow\beta \cdot B\alpha, a]\) does not, but closure ensures that all the rhs’s for B are in \( s_x \)
Ex ACTION and GOTO tables

1. $S \rightarrow E$
2. $E \rightarrow T+E$
3. $T \rightarrow id$
4. $T \rightarrow id$

```
ACTION | GOTO
---|---
id | + | $ | E | T
S0 | s3 |  |  |  |
S1 |  |  | acc |  |  |
S2 | s4 |  | r3 |  |  |
S3 |  | r4 |  | r4 |  |
S4 | s3 |  |  |  | 5 | 2 |
S5 |  |  |  | r2 |  |  |
```

```
S0
[S \rightarrow \cdot E, $]
[E \rightarrow \cdot T+E, $]
[E \rightarrow \cdot T, $]
[T \rightarrow \cdot id, +]
[T \rightarrow \cdot id, $]

E \downarrow
S1
[S \rightarrow E \cdot, $]

S2
[T \rightarrow id \cdot, +]

S3

T

S4
[E \rightarrow T + \cdot E, $]
[E \rightarrow \cdot T+E, $]
[E \rightarrow \cdot T, $]
[T \rightarrow \cdot id, +]
[T \rightarrow \cdot id, $]

T

S5
[E \rightarrow T + E \cdot, $]

id

id

id

E \downarrow

E\downarrow
```
Ex ACTION and GOTO tables

1. $S \rightarrow E$
2. $E \rightarrow T+E$
3. $T \rightarrow id$

<table>
<thead>
<tr>
<th></th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>s3</td>
<td>E</td>
</tr>
<tr>
<td>S1</td>
<td>acc</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>s4</td>
<td>r3</td>
</tr>
<tr>
<td>S3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>S4</td>
<td>s3</td>
<td>5</td>
</tr>
<tr>
<td>S5</td>
<td>r2</td>
<td>2</td>
</tr>
</tbody>
</table>

Entries for shift:

- $S \rightarrow • E, \$, $E \rightarrow • T+E, \$
- $E \rightarrow • T, \$
- $T \rightarrow • id, +$
- $T \rightarrow • id, \$

- $E \rightarrow T + • E, \$
- $E \rightarrow • T+E, \$
- $E \rightarrow • T, \$
- $T \rightarrow • id, +$
- $T \rightarrow • id, \$

- $E \rightarrow T + E •, \$
- $E \rightarrow T + • E, \$
- $E \rightarrow • T+ E, \$
- $E \rightarrow • T, \$
- $T \rightarrow • id, +$
- $T \rightarrow • id, \$

- $E \rightarrow T + E •, \$
Ex ACTION and GOTO tables

1. $S \rightarrow E$
2. $E \rightarrow T+E$
3. $| T$
4. $T \rightarrow id$

ACTION and GOTO tables:

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>$id$</td>
<td>$+$</td>
</tr>
<tr>
<td>$S0$</td>
<td>$s3$</td>
</tr>
<tr>
<td>$S1$</td>
<td></td>
</tr>
<tr>
<td>$S2$</td>
<td>$s4$</td>
</tr>
<tr>
<td>$S3$</td>
<td></td>
</tr>
<tr>
<td>$S4$</td>
<td>$s3$</td>
</tr>
<tr>
<td>$S5$</td>
<td></td>
</tr>
</tbody>
</table>

Entry for accept
Ex ACTION and GOTO tables

1. S → E
2. E → T+E
3. | T
4. T → id

Entries for reduce

<table>
<thead>
<tr>
<th></th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>$</td>
</tr>
<tr>
<td>S0</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td>acc</td>
</tr>
<tr>
<td>S2</td>
<td>s4</td>
<td>r3</td>
</tr>
<tr>
<td>S3</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>S4</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>S5</td>
<td></td>
<td>r2</td>
</tr>
</tbody>
</table>

[S → • E, $]  
[E → • T+E, $]  
[E → • T, $]  
[T → • id, +]  
[T → • id, $]

[S → • E, $]  
[T → • id, $]  
[E → T • +E, $]  
[E → T • , $]

[S → E •, $]  
[T → id • , +]  
[T → id • , $]

[S → E •, $]  
[T → id • , $]  
[E → T + • E, $]

[S → • E, $]  
[T → • id, $]  
[E → T •, $]  
+E

T

id

id
Ex ACTION and GOTO tables

1. $S \rightarrow E$
2. $E \rightarrow T+E$
3. $| T$
4. $T \rightarrow id$

Entries for GOTO

- $S \rightarrow E •$
- $E \rightarrow T • +E, \$
- $E \rightarrow T •$
- $T \rightarrow • id, +$
- $T \rightarrow • id, \$

ACTION and GOTO tables:

<table>
<thead>
<tr>
<th></th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>+</td>
<td>$E$</td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>acc</td>
<td>2</td>
</tr>
<tr>
<td>s4</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

[S $\rightarrow \bullet E, \$]
[E $\rightarrow \bullet T+E, \$]
[E $\rightarrow \bullet T, \$]
[T $\rightarrow \bullet id, +$]
[T $\rightarrow \bullet id, \$]

[S $\rightarrow E \bullet, \$]
[T $\rightarrow id \bullet, +$]
[T $\rightarrow id \bullet, \$]

[E $\rightarrow T + \bullet E, \$]
[E $\rightarrow \bullet T+E, \$]
[E $\rightarrow \bullet T, \$]
[T $\rightarrow \bullet id, +$]
[T $\rightarrow \bullet id, \$]
What can go wrong?

• What if set $s$ contains $[A \rightarrow \beta \cdot a \gamma, b]$ and $[B \rightarrow \beta \cdot a]$?
  ▪ First item generates “shift”, second generates “reduce”
  ▪ Both define $\text{ACTION}[s,a]$ — cannot do both actions
  ▪ This is a shift/reduce conflict

• What if set $s$ contains $[A \rightarrow \gamma \cdot a]$ and $[B \rightarrow \gamma \cdot a]$?
  ▪ Each generates “reduce”, but with a different production
  ▪ Both define $\text{ACTION}[s,a]$ — cannot do both reductions
  ▪ This is called a reduce/reduce conflict

• In either case, the grammar is not LR(1)
Shift/reduce conflict

- Associativity unspecified
  - Ambiguous grammars always have conflicts
  - But, some non-ambiguous grammars also have conflicts
Solving conflicts

• Refactor grammar
• Specify operator precedence and associativity

### Lots of details here
- See “12.4.2 Declarations” at

### When comparing operator on stack with lookahead
- Shift if lookahead has higher prec OR same prec, right assoc
- Reduce if lookahead has lower prec OR same prec, left assoc

### Can use smaller, simpler (ambiguous) grammars
- Like the one we just saw

```ocaml
%left PLUS MINUS  (* lowest precedence *)
%left TIMES DIV    (* medium precedence *)
%nonassoc UMINUS   (* highest precedence *)
```
Left vs. right recursion

• Right recursion
  ▪ Required for termination in top-down parsers
  ▪ Produces right-associative operators

• Left recursion
  ▪ Works fine in bottom-up parsers
  ▪ Limits required stack space
  ▪ Produces left-associative operators

• Rule of thumb
  ▪ Left recursion for bottom-up parsers
  ▪ Right recursion for top-down parsers
Reduce/reduce conflict (1)

• Often these conflicts suggest a serious problem
  ▪ Here, there’s a deep ambiguity
### Reduce/reduce conflict (2)

```plaintext
%token <int> INT
%token EOL PLUS LPAREN RPAREN
%start main /* the entry point */
%type <int> main
%
main:  
| expr EOL { $1 }

expr:  
| term1 { $1 }
| term1 PLUS PLUS expr { $1 + $4 }
| term2 PLUS expr { $1 + $3 }

term1 :  
| INT { $1 }
| LPAREN expr RPAREN { $2 }

term2 :  
| INT { $1 }
```

- Grammar not ambiguous, but not enough lookahead to distinguish last two `expr` productions
Shrinking the tables

• Combine terminals
  - E.g., number and identifier, or + and -, or * and /
  - Directly removes a column, may remove a row

• Combine rows or columns (*table compression*)
  - Implement identical rows once and remap states
  - Requires extra indirection on each lookup
  - Use separate mapping for ACTION and for GOTO

• Use another construction algorithm
  - LALR(1) used by ocamlyacc
LALR(1) parser

• Define the core of a set of LR(1) items as
  ▪ Set of LR(0) items derived by ignoring lookahead symbols

\[
\begin{align*}
[E &\rightarrow a \cdot, b] \\
[A &\rightarrow a \cdot, c] \\
[E &\rightarrow a \cdot] \\
[A &\rightarrow a \cdot]
\end{align*}
\]

LR(1) state  Core

• LALR(1) parser merges two states if they have the same core

• Result
  ▪ Potentially much smaller set of states
  ▪ May introduce reduce/reduce conflicts
  ▪ Will not introduce shift/reduce conflicts
LALR(1) example

- Introduces reduce/reduce conflict
  - Can reduce either $E \rightarrow a$ or $A \rightarrow ba$ for lookahead = $b$

LR(1) states

- $[E \rightarrow a \cdot, b]$
- $[A \rightarrow ba \cdot, c]$
- $[E \rightarrow a \cdot, d]$
- $[A \rightarrow ba \cdot, b]$

Merged state

- $[E \rightarrow a \cdot, b]$
- $[A \rightarrow ba \cdot, c]$
- $[E \rightarrow a \cdot, d]$
- $[A \rightarrow ba \cdot, b]$
LALR(1) vs. LR(1)

• Example grammar

\[
\begin{align*}
S' & \rightarrow S \\
S & \rightarrow aAd | bBd | aBe | bAe \\
A & \rightarrow c \\
B & \rightarrow c
\end{align*}
\]

• LR(0) ?

• LR(1) ?

• LALR(1) ?
LR(k) Parsers

• Properties
  ▪ Strictly more powerful than LL(k) parsers
  ▪ Most general non-backtracking shift-reduce parser
  ▪ Detects error as soon as possible in left-to-right scan of input
    - Contents of stack are viable prefixes
      - Possible for remaining input to lead to successful parse
Error handling (lexing)

• What happens when input not handled by any lexing rule?
  ▪ An exception gets raised
  ▪ Better to provide more information, e.g.,

    ```haskell
    rule token = parse
    ...
    | _ as lxm { Printf.printf "Illegal character %c" lxm;
               failwith "Bad input" }
    ```

• Even better, keep track of line numbers
  ▪ Store in a global-ish variable (oh no!)
  ▪ Increment as a side effect whenever `\n` recognized
Error handling (parsing)

• What happens when parsing a string not in the grammar?
  - Reject the input
  - Do we keep going, parsing more characters?
    - May cause a cascade of error messages
    - Could be more useful to programmer, if they don’t need to stop at the first error message (what do you do, in practice?)

• Ocamlyacc includes a basic error recovery mechanism
  - Special token `error` may appear in rhs of production
  - Matches erroneous input, allowing recovery
Error example (1)

```plaintext
...  
expr:  
  | term                { $1 }  
  | expr PLUS term      { $1 + $3 }  
  | error               { Printf.printf "invalid expression"; 0 }  
term: ...
```

- If unexpected input appears while trying to match `expr`, match token to `error`
  - Effectively treats token as if it is produced from `expr`
  - Triggers error action
Error example (2)

If unexpected input appears while trying to match `term`, match tokens to `error`

- Pop every state off the stack until `LPAREN` on top
- Scan tokens up to `RPAREN`, and discard those, also
- Then match `error` production

```
term:
  | INT { $1 }
  | LPAREN expr RPAREN { $2 }
  | LPAREN error RPAREN { printf "Syntax error!\n"; 0}
```
Error recovery in practice

• A very hard thing to get right!
  ▪ Necessarily involves guessing at what malformed inputs you may see

• How useful is recovery?
  ▪ Compilers are very fast today, so not so bad to stop at first error message, fix it, and go on
  ▪ On the other hand, that does involve some delay

• Perhaps the most important feature is good error messages
  ▪ Error recovery features useful for this, as well
  ▪ Some compilers are better at this than others
OCamlyacc tip

• Setting OCAMLRUNPARAM=p will cause the parsing steps to be printed out as the parser runs.
• (And setting OCAMLRUNPARAM=b will tell OCaml to print a stack backtrace for any thrown exceptions.)
Real programming languages

• Essentially all real programming languages don’t quite work with parser generators
  ▪ Even Java is not quite LALR(1)

• Thus, real implementations play tricks with parsing actions to resolve conflicts

• In-class exercise: C typedefs and identifier declarations/definitions
Additional Parsing Technologies

• For a long time, parsing was a “dead” field
  ▪ Considered solved a long time ago
• Recently, people have come back to it
  ▪ LALR parsing can have unnecessary parsing conflicts
  ▪ LALR parsing tradeoffs more important when computers were slower and memory was smaller
• Many recent new (or new-old) parsing techniques
  ▪ GLR — generalized LR parsing, for ambiguous grammars
  ▪ LL(*) — ANTLR
  ▪ Packrat parsing — for parsing expression grammars
  ▪ etc...
• The input syntax to many of these looks like yacc/lex
Designing language syntax

• Idea 1: Make it look like other, popular languages
  ▪ Java did this (OO with C syntax)

• Idea 2: Make it look like the domain
  ▪ There may be well-established notation in the domain (e.g., mathematics)
  ▪ Domain experts already know that notation

• Idea 3: Measure design choices
  ▪ E.g., ask users to perform programming (or related) task with various choices of syntax, evaluate performance, survey them on understanding
    - This is very hard to do!

• Idea 4: Make your users adapt
  ▪ People are really good at learning...