Today’s Topics

Last Time
- Top-down parsers - predictive parsing, backtracking, recursive descent, LL parsers, relation to S/SL

This Time
- Constructing parsers in SL
- Syntax error recovery and repair
Parsing Using SL

• As we have observed, an **SL** program (an **S/SL** program with no semantic mechanisms) is a **context-free parser**

• The output of an **SL** parser is a stream of **semantic tokens** - that is, input tokens for the semantic analyzer

• Expressions are translated by the parser to **postfix** notation using **SL** rules, one for each level of **precedence**

\[ 1 + 4 \times ( 7 \times 6 + 9 ) \rightarrow 1 \; 4 \; 7 \; 6 \; * \; 9 \; + \; * \; + \]

• Postfix is a convenient linear notation for the output **parse tree** - all precedences have been resolved, and the **order of operations** has been determined and is represented directly

• Subsequent phases (**semantic analysis, code generation**) need not be bothered with precedence or associativity
Understanding Postfix

• Postfix is simply a convenient linear notation for a **parse tree**

• Constructing the represented parse tree consists of reading from **left to right**, rotating the **operators** above the previous two items

• Example:

```
  1 4 7 6 * 9 + * +
  1 4  * 9 + * +
    7 6
  1 4 + * +
      * 9
        7 6
  1 + +
     4
      * 9
        7 6
```
SL Parsing - Expressions

• For precedences:
  (C, Java, Pascal)

  1. unary –     highest (most binding)
  2. * /          lowest (least binding)
  3. + –

Example:

\[ 1 + - 3 \times 2 \rightarrow 1 \ 3 \ \text{sMinus} \ 2 \ \text{sMult} \ \text{sAdd} \]
SL Parsing - Expressions

• For precedences:
  1. * /  
  2. unary –  
  3. + –  

Example:

\[ 1+(-3\times2) \rightarrow 1\ 3\ 2\ \text{smult}\ \text{sminus}\ \text{sadd} \]
Role of the SL Parser

- The Parser’s job is to:
  1. check syntax to see that it is valid,
  2. resolve precedence and order of operations (associativity),
  3. recognize and explicitly relate structure of statements, blocks and scopes

- Each kind of Parser achieves these aims in different ways

- The SL Parser does the first of these automatically using S/SL syntax error detection (more on this later)

- It does the second by converting expressions to their parse tree in postfix form

- It does the third by recognizing structural relationships and explicitly marking them using semantic tokens
Syntax Error Recovery

- A Parser must *detect* syntax errors - but it must also:
  - *Isolate* them, so that the user knows *where* the mistake is, and
  - *Recover* from them, so that it can *continue* parsing and check the rest of the program

- Parsing methods vary in their ability to do each of these well - all methods can *detect* syntax errors (otherwise they wouldn’t be parsers)

- *Isolating* them (pointing out where the actual fault is) is more difficult, and *recovering* from them without causing more (*phantom*) errors is even more difficult
Syntax Error Recovery

• Most parsing algorithms provide some general recovery strategy

• Basic problem:
  • When syntax error is detected, (a mismatch between the next input token and what’s expected by the grammar),
  • Resynchronize the state of the parse and the next input token with a minimum number of actions and the minimum probability of further errors

• Effective syntax error recovery and repair is a big area of research, with many complex algorithms

• Varying degrees of success, ranging from virtually useless (with many cascading “phantom” errors), to virtually perfect, with little or no cascading

• Success varies depending on the particular error - most recovery algorithms have one or more “pathological cases”, where they do very badly
SL Syntax Error Recovery

• The built-in syntax error recovery used by SL (by Barnard, 1981) is based on using end-of-statement markers (e.g. ‘;’) as synchronization points for the parse.

• This is a common idea in many syntactic error recovery strategies - e.g. COBOL compilers typically skip to the next statement keyword that starts a statement.
SL Syntax Error Recovery

- In **SL**, a syntax error is detected when the **expected** input token (in either an explicit **input token** action or an **input choice** action) does not match the next input token.

- The steps in recovery are:
  1. Suspend input token acceptance (enter **recovery state**)
  2. In recovery state, continue to walk S/SL table, **pretending** all expected input tokens are matched but **accepting none**, until we expect an **end-of-statement** token.
  3. Flush (discard) tokens from input until an **end-of-statement** token.
  4. Exit recovery state and continue as if nothing had happened.
SL Syntax Error Recovery - Example

VarDeclaration:
  pIdentifier
  .sIdentifier
  ;;
@TypeBody

TypeBody:
  [ ...
    | 'array':
    | 'file':
    | 'integer':
      *:
        @SimpleType

SimpleType:
  [ ...
    | pIdentifier:
      *
        @Constant
        '..' .sRange
        @Constant
  ];

Constant:
  pInteger
  .sIntegerLiteral;

Input:
  var x = integer;

Recovery/repair:
  var x : 1..1 ;
SL Syntax Error Recovery - Input Choices

- If the error occurs during an input choice (i.e. the next input token does not match any alternative, and the choice has no default) then SL treats it as a mismatch of the first alternative

- So for recovery purposes (only),

  Constant:
  
  |   | pIdentifier: .sIdentifier
  | pInteger: .sInteger
  |

  Acts like:

  Constant:
  
  |   | pInteger: .sInteger
  |   | pIdentifier .sIdentifier
  |

  In other words, the error is treated as a mismatch of the first alternative, and then recovery proceeds as usual

  And if we are in recovery state, the first alternative is always taken
Recovery Loops

- SL recovery state walks the SL program looking for an expected end-of-statement token, pretending all input is matched.

- When coding SL we have to be aware of the possibility of recovery - otherwise the path followed in recovery state may never reach and expected end-of-statement, and we have an infinite loop!

  \[
  \text{ConstantList:}
  \]
  \[
  \left(\left(\begin{array}{l}
    \text{atConstant} \\
    \{\\n      | \text{'};\\n      > \\
      | \text{::*}\\n      \text{','} @\text{Constant}\\n    \}\\n  \right)\right);\\n  \]

  - Each time around the loop, the default (\text{::*}) is taken.
  - Must structure SL rules such that recovery will always terminate.

- If input is (1, 3; 2) → infinite loop
Recovery Loop Resolution

- A simple rule-of-thumb makes it easy to be sure recovery will always terminate

- Always make the exiting alternative of an SL loop either:
  (a) the default (otherwise) alternative, or
  (b) the first alternative of a choice without a default

ConstantList: ConstantList:
  ‘(‘
  @Constant
  {[ [ ‘,’:
      @Constant
      | ‘,’:
      @Constant
      | ‘,’:
      @Constant
      | ‘,’:
      @Constant
    ]} ‘)’;

- If input is (1, 3; 2)
  → exits the loop as soon as enters recovery state
What if no Semicolons?

• Easier syntax error recovery was one of the original motivating factors behind the use of semicolons in programming languages.

• Some languages, such as Euclid and Turing, do not have semicolons or other any other end-of-statement tokens (but are still free-format, that is, end-of-lines are not required).

• So how can the SL parser resynchronize after a syntax error?

• Observation: Programmers do not format their code in arbitrary ways - newlines (return or enter keys) tend to be typed at the ends of expressions and statements - therefore we can use the actual newline characters as synchronization points.

• Of course this will not always work perfectly - but quite often in Computer Science we tend to overgeneralize and worry about the worst case when it’s not reasonable - in 99% of cases this works.

• If there happens to be a misplaced newline in the middle of a statement and there is an error, the only penalty will be a single extra “cascaded” syntax error.
SL Syntax Error Recovery - No Semicolons

• Insert special *new line* (‘<nl>’) tokens into the grammar at ends of statements and other forms where semicolons “would be”

• Example:

```plaintext
VarDeclarations:
  pIdentifier .sIdentifier
  ':'
@TypeBody
  '<nl>':'
```

• SL Parser modifications to handle this:

1. Mismatches between *expected* newline tokens and input tokens are ignored (not syntax errors because newlines not required in the language)

2. Newline tokens are used as synchronization points:
   (a) Recovery state walks to the next expected newline token
   (b) Input is flushed to the next newline in the input
Summary

SL Parsers

• Parsers in SL output a postfix token stream which represents a parse tree in linear form.
• SL parsers encode precedence and associativity using cascaded nonterminals as in BNF.
• Syntax error recovery is an important and difficult problem.
• SL has a simple built-in syntax error recovery strategy based on end-of-statement markers such as semicolon.
• SL parsers must be coded to take into account the recovery strategy in order to avoid “recovery loops”.

Next Week

• Programming language semantics
• Runtime model of execution

• Assignment #1 DUE - Wednesday, Feb 4 (before next class)