Today

- Quiz 2
- Syntax Analysis: Parsing

Assignments

- HW-2, R-2 due
- HW-3 (extended), R-3 (out, next Thursday)
Parsing: An example grammar

Simple list of assignment statements

\[
stmt\_list \rightarrow stmt \mid stmt \ ';$' \ stmt\_list
\]

\[
stmt \rightarrow var \ '=$' \ expr
\]

\[
var \rightarrow 'A' \mid 'B' \mid 'C'
\]

\[
expr \rightarrow var \mid var \ '+' \ var \mid var \ '-' \ var
\]

– quotes used here to help distinguish terminals from non-terminals
– Note: many possible grammars for our language!

We can use grammars to generate strings (derivations)

1. choose a production (start symbol on left-hand side)
2. replace with right-hand side
3. pick a non-terminal \( N \) and production with \( N \) on left side
4. replace \( N \) with production's right-hand side
5. continue until only terminals remain
Example derivation of “A = B + C; B = A”

\[\begin{align*}
stmt\_list & \Rightarrow stmt \ ; \ stmt\_list \\
& \Rightarrow var = expr \ ; \ stmt\_list \\
& \Rightarrow A = expr \ ; \ stmt\_list \\
& \Rightarrow A = var + var \ ; \ stmt \\
& \Rightarrow A = B + var \ ; \ stmt\_list \\
& \Rightarrow A = B + C \ ; \ stmt\_list \\
& \Rightarrow A = B + C \ ; \ stmt \\
& \Rightarrow A = B + C \ ; \ var = expr \\
& \Rightarrow A = B + C \ ; \ B = expr \\
& \Rightarrow A = B + C \ ; \ B = var \\
& \Rightarrow A = B + C \ ; \ B = C
\end{align*}\]

- This is a “left-most” derivation
  - derived the string by replacing left-most non-terminals

- The opposite is a “right-most” derivation

\[\begin{align*}
stmt\_list & \Rightarrow stmt \ ; \ stmt\_list \\
& \Rightarrow var = expr \ ; \ stmt\_list \\
& \Rightarrow var = expr \ ; \ stmt \\
& \Rightarrow var = expr \ ; \ var = expr \\
& \Rightarrow \ldots
\end{align*}\]

- Sometimes write \(\Rightarrow^*\) for a multi-step derivation
  - e.g.: \(stmt\_list \Rightarrow^* var = expr \ ; \ var = expr\)
Exercise: Give a left-most derivation of $A = B$

Exercise: Give a right-most derivation of $A = B$

Exercise: Give a derivation that is neither right- nor left-most for $A = B$
Derivations can also be written as “parse trees”

- Using the previous example derivation of “A = B + C; B = A”
Parsing

- A context free grammar is a “generator”
- Whereas a parser is a “recognizer”
  - given a token stream
  - determine if the stream is a derivation of the grammar
- Typically a parser also builds an Abstract Syntax Tree (AST)

We’ll look at $LL(k)$ parsers

- read from **left-to-right**, performing a **left-most** derivation
- parses **top down** (parse tree from the root down)
- at most $k$ look ahead symbols (more later)

Consider this (modified) rule:

\[ stmt \rightarrow ( \text{‘A’} \mid \text{‘B’} \mid \text{‘C’} ) \text{ ‘=’ expr} \]

Assuming the parser knows this rule is to be applied ...

1. it calls lexer’s `next_token`
2. it checks if it is a literal "A", "B", or "C"
3. it calls lexer’s `next_token`
4. it checks that it is an `ASSIGN` token
5. and so on until it finishes the `stmt` rule

- the parser produces an **error** if it finds a token it isn’t expecting
**Tips for \( LL(k) \)**

**Watch out for left recursion!**

R1: \( e \to n \)

R2: \( e \to e + n \)

Q: how far do we need to look ahead for “5 + 4 + 3”?

- we have to go to the end of the expression …
- even though we’re doing a left-most derivation!

1. Looking at 5 (1 lookahead), we don’t know whether to apply R1 or R2 (\( n \Rightarrow 5 \) and \( e \Rightarrow 5 \))
2. But to know if R2 should be applied, we need to know if the string ***ends*** in “+ n”
3. This means we have to read the entire string to know which rule to apply
4. If the string is longer than our fixed size \( k \), then we are stuck!

**One solution**

\[
e \to n + e | n
\]

Q: How many look aheads needed? ... 2 (see “left factoring”)

**Can rewrite left recursion to be in \( LL(k) \)...**

\[
e \to n e' \\
e' \to + n e' | \epsilon
\]

Q: now how far do we need to look ahead for “5 + 4 + 3”?
The above example involved **immediate** (direct) left recursion

A grammar can also have **indirect** left recursion

\[
s \rightarrow t \ a \ | \ a \\
t \rightarrow s \ b \ | \ b
\]

- allows derivations: \( s \Rightarrow t \ a \Rightarrow s \ b \ a \)
- having strings of the form: \( a, ba, aba, baba, ababa, \ldots \)

Example rewriting for this grammar

- By replacing LHS of \( t \) in \( s \), we get:
  
  \[
  s \rightarrow s \ b \ a \ | \ b \ a \ | \ a
  \]

Now we can rewrite the above

\[
s \rightarrow a \ s' \ | \ ba \ s' \\
s' \rightarrow ba \ s' \ | \ \epsilon
\]
Sometimes we need to left factor ... 

\[ e \rightarrow \text{if } b \text{ then } s \mid \text{if } b \text{ then } s \text{ else } s \]

- here the first and second choice have a common \text{prefix}
- this means more look-ahead tokens than needed

After left factoring ...

\[ e \rightarrow \text{if } b \text{ then } s \ r \]
\[ r \rightarrow \text{else } s \mid \epsilon \]
What out for ambiguous grammars!

\[ e \rightarrow id \mid p \]
\[ p \rightarrow [ id ] \mid id \]

- here there are multiple (left-most) ways to generate an id
  \[ e \Rightarrow id \Rightarrow x \]
  \[ e \Rightarrow p \Rightarrow id \Rightarrow x \]

- the problem is that these produce different parse trees
- and thus, may have different language interpretations (more later)
Q: Can you spot any of these problems in our example?

\[
\begin{align*}
stmt_list & \rightarrow stmt \mid stmt \ ';$' \ stmt_list \\
stmt & \rightarrow var \ '=$' \ expr \\
var & \rightarrow 'A' \mid 'B' \mid 'C' \\
expr & \rightarrow var \mid var \ '+' \ var \mid var \ '-' \ var
\end{align*}
\]

Q: Is it left-recursive? No
Q: Can it be left factored? Yes
Q: Is it ambiguous? No
Q: How many look ahead tokens needed? 6 for \textit{stmt\_list} (A=B+C;\ldots)
A more standard way to write the grammar (with problems fixed)

\[
\begin{align*}
\text{stmt} &\rightarrow \text{var} \,'='\, \text{expr} \,\text{stmt} \,\text{list} \\
\text{stmt} \,\text{list} &\rightarrow \,';'\, \text{stmt} \,\text{list} \,\epsilon \\
\text{var} &\rightarrow \,'A' \mid \,'B' \mid \,'C' \\
\text{expr} &\rightarrow \text{var} \,\text{expr} \,\text{tail} \\
\text{expr} \,\text{tail} &\rightarrow \,'+'\, \text{var} \mid \,'-'\, \text{var} \mid \epsilon
\end{align*}
\]

Note that for parsing, it is convenient to rewrite using token types ...

\[
\begin{align*}
\text{stmt} \,\text{list} &\rightarrow \text{VAR ASSIGN} \,\text{expr} \,\text{stmt} \,\text{list} \\
\text{stmt} \,\text{list} \,\text{tail} &\rightarrow \text{SEMICOLON} \,\text{stmt} \,\text{list} \mid \epsilon \\
\text{expr} &\rightarrow \text{VAR} \,\text{expr} \,\text{tail} \\
\text{expr} \,\text{tail} &\rightarrow \text{PLUS} \,\text{VAR} \mid \text{MINUS} \,\text{VAR} \mid \epsilon
\end{align*}
\]