Today

• Intro to parsing

This Week

• EX-3 available
• HW-2 due
• HW-3 out
• Quiz-2 Thursday (lexer, grammars, derivations)
Parsing: An example grammar

Simple list of assignment statements

\[
\begin{align*}
<\text{stmt_list}> & ::= <\text{stmt}> \mid <\text{stmt}> ';' <\text{stmt_list}> \\
<\text{stmt}> & ::= <\text{var}> '=' <\text{expr}> \\
<\text{var}> & ::= 'A' \mid 'B' \mid 'C' \\
<\text{expr}> & ::= <\text{var}> \mid <\text{var}> '+' <\text{var}> \mid <\text{var}> '-' <\text{var}>
\end{align*}
\]

Note: many possible grammars for this language!

We can use grammars to generate strings (derivations)

1. choose a rule (e.g., with start symbol on left-hand side)
2. replace with right-hand side (of rule)
3. pick a non-terminal \( N \) and rule with \( N \) on left side
4. replace \( N \) with rule’s right-hand side
5. repeat from 3 until only terminals remain

Whereas \( \rightarrow \) (or \( ::= \)) denotes a rule, \( \Rightarrow \) denotes a derivation
Example derivation of “A = B + C; B = A”

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

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

- The opposite is a “right-most” derivation
Derivations can also be written as “parse trees”

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

Two real-world PL grammar examples:

- Python3 grammar: docs.python.org/3/reference/grammar.html
- Java 15 grammar: docs.oracle.com/javase/specs/jls/se15/html/index.html
Parsing

- A context free grammar (derivation) is a "generator"
- Whereas a parser is a "recognizer"
  - given a token stream
  - determine if the stream is a derivation of the grammar
- A parser also (typically) 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 these (modified) rules:

\[<\text{stmt}> ::= 'A' '=' <\text{expr}>\]
\[<\text{stmt}> ::= 'B' '=' <\text{expr}>\]
\[<\text{stmt}> ::= 'C' '=' <\text{expr}>\]

Assuming the parser knows $<\text{stmt}>$ is to be applied ... 

1. calls lexer’s `nextToken`
2. checks if it is a literal "A", "B", or "C", picking the corresponding rule
3. calls lexer’s `next_token`
4. checks that it is an `ASSIGN` token
5. and so on until it finishes the $<\text{stmt}>$ rule

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

Watch out for left recursion!

R1: $e \rightarrow n$
R2: $e \rightarrow e + n$

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
2. To decide R2, 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

\[ e \rightarrow n + e \mid n \]

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

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

\[ e \rightarrow n \ e' \]
\[ e' \rightarrow + \ n \ e' \mid \epsilon \]

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

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

- 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 RHS of \( t \) in \( s \), we get:
  \[ s \rightarrow s \ b \ a \mid b \ a \mid a \]

Now we can rewrite the above

\[ s \rightarrow a \ s' \mid ba \ s' \]
\[ s' \rightarrow ba \ s' \mid \epsilon \]
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 prefix
- this generally means more look-ahead tokens than needed
- in this example, unless \( b \) and \( s \) are of fixed sized, there’s no fixed \( k \):

After left factoring ...

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

\[ r \rightarrow \text{else } s \mid \epsilon \]

- Note that this is now \( LL(1) \)
What out for *ambiguous* grammars!

\[ e \rightarrow id \mid p \]

\[ p \rightarrow [ \text{id} ] \mid \text{id} \]

- 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)