Recall syntax directed translation (SDT)

SDT = translating from a sequence of tokens into a sequence of actions/other form, based on underlying syntax

To define a syntax-directed translation

- augment CFG with translation rules
  - define translation of LHS non-terminal as a function of: constants, translations of RHS non-terminals, values of terminals (tokens) on RHS

To translate a sequence of tokens using SDT (conceptually)

- build parse tree
- use translation rules to compute translation of each non-terminal (bottom-up)
- translation of sequence of tokens = translation of parse tree's root non-terminal

Examples

- translate tokenized stream to an integer value
- translate tokenized stream to a string

For parsing, we'll need to translate tokenized stream to abstract-syntax tree (AST)
Abstract syntax trees

AST = condensed form of parse tree

- operators are internal nodes (not leaves)
- chains of productions are collapsed
- lists are flattened
- syntactic details are omitted

Example: \((5 + 2) \times 8\)
Example 2

CFG
expr → expr + term
| term

term → term * factor
| factor

factor → INTLIT
| ( expr )

AST for parsing

\[ (1+2) \times (3+4) \times 5 + 6 \]

In practice, token stream → AST

AST - captures essential structure
- easier to work with
AST implementation

\[ \text{expr} \to \text{expr} + \text{term} \]

\[ \text{exp}_1 \cdot \text{trans} = \text{MkPlusNode}(\text{exp}_2 \cdot \text{trans}, \text{term} \cdot \text{trans}) \]

- Define class for each kind of AST node
  - Create a new node object in some rules
    - new node object is value of LHS \cdot \text{trans}
    - fields of node object come from translations of RHS non-terminals

Given 1 + 2

Parse tree

\[
\begin{align*}
\text{expr} & \to \text{term} + \text{term} \\
\text{term} & \to \text{factor} \cdot \text{factor} \\
\text{factor} & \to \text{INTLIT} \\
\end{align*}
\]

AST

\[
\begin{align*}
\text{PlusNode} & \quad \square \\
\text{InrNode} 1 & \quad \square \\
\text{InrNode} 2 & \quad \square \\
\end{align*}
\]

Need class hierarchy & make these subclasses of ExpNode

Class hierarchy:

- \text{ExpNode}
  - \text{InrNode}
    - \text{InrNode left}
    - \text{InrNode right}
  - \text{PlusNode}
  - \text{IntNode}
    - \text{int value}

Put into ExpNode
Translation rules to build ASTs for expressions

CFG
expr → expr + term
  | term

term → term * factor
  | factor

factor → INTLIT
  | ( expr )

Example: 1 + 2

Translation rules
expr.trans = new PlusNode(expr.trans, term.trans)
expr.trans = term.trans

term.trans = new TimesNode(term.trans, factor.trans)

factor.trans = new IntNode(INTLIT.value)

factor.trans = expr.trans
ASTs for non-expressions

Example

```cpp
void foo(int x, int y) {
    if (x == y) {
        return;
    }
    while (x < y) {
        cout << "hello";
        x = x + 1;
    }
    return;
}
```
ASTs for lists

CFG

idList → idList COMMA ID
| ID

Want AST to be

Parse tree

translation adds IdNode for "z" to end of list

translation adds IdNode for "y" to end of list

translation is a new list with IdNode for "x"
The bigger picture

Scanner

- **Language abstraction**: regex
- **Output**: token stream
- **Tool**: JLex
- **Implementation**: interpret DFA using table (for $\delta$), recording most_recent_accepted_position & most_recent_token

Parser

- **Language abstraction**: CFG
- **Output**: $\text{AST}$ (by way of syntax-directed translation)
- **Tool**: JavaCup (next time)
- **Implementation**: (next week)