CS 536 Announcements for Monday, February 19, 2024

Programming Assignment 2
• due Tuesday, February 20

Last Time
• syntax-directed translation
• abstract syntax trees

Today
• implementing ASTs

Next Time
• Java CUP

SDT review

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 \( z, \"\" \)
    • 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

For parsing, we'll need to translate tokenized stream to abstract-syntactic tree (AST)
Example $\left( (1+2) \times (3+4) \right) \times 5 + 6$

AST for parsing

We’ve been showing the translation in two steps:

$\text{token stream} \rightarrow \text{parse tree} \rightarrow \text{AST}$
then throw away parse tree

In practice we’ll do

$\text{token stream} \rightarrow \text{AST}$

Why have an AST?

- captures essential structure
- easier to work with
AST implementation

Define a class for each kind of AST node

Create a new node object in some rules
- new node object is the value of LHS.trans
- fields of node object come from translations of RHS non-terminals

Given 1+2
Parse tree

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{term} \\
\text{expr}_1 \cdot \text{trans} & = \text{mkPlusNode} (\text{expr}_2 \cdot \text{trans}, \text{term} \cdot \text{trans})
\end{align*}
\]

Need class hierarchy & make these subclases of ExpNode

class PlusNode extends ExpNode

class IntNode extends ExpNode

class IntNode Left:
int value
Translation rules to build ASTs for expressions

CFG

expr → expr + term
| term

term → term * factor
| factor

factor → INTLIT
| ( expr )

Translation rules

expr₁.trans = new PlusNode(expr₂.trans, term₁.trans)
expr.trans = term₁.trans

term₁.trans = new TimesNode(term₂.trans, factor₁.trans)
term.trans = factor₁.trans

factor₁.trans = new INLIT(INTLIT.value)
factor.trans = expr₁.trans

Example 1 + 2

expr
expr + term
| term
| factor
| INLIT (1)

PlusNode

INLIT (2)

INLIT (1)
Example

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

ASTs for non-expressions
ASTs for lists

CFG

\[
idList \rightarrow \text{idList COMMA ID} \\
| \text{ID}
\]

Want AST to be

\[
\text{Input: } x, y, z
\]

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**: regular expressions
- **Output**: token stream
- **Tool**: JLex
- **Implementation**: interpret DFA using table (for δ), recording most_recent_accepted_position & most_recent_token

Parser
- **Language abstraction**: **CFG**
- **Output**: AST (by way of a syntax-directed translation)
- **Tool**: JavaCup
- **Implementation**: ???.next time

[Diagram showing the process of regex action through JLex to scanner tokens, and then through JavaCup to parser AST]