Announcements

Working in pairs is only allowed for programming assignments and not for homework problems

H3 has been posted
Syntax Directed Translation
CFGs so Far

CFGs for Language *Definition*

– The CFGs we’ve discussed can generate/define languages of valid strings
– So far, we **start** by building a parse tree and **end** with some valid string

CFGs for Language *Recognition*

– Start with a string and end with a parse tree for it
CFGs for Parsing

Language Recognition isn’t enough for a parser
  – We also want to translate the sequence
Parsing is a special case of Syntax-Directed Translation
  – Translate a sequence of tokens into a sequence of actions
Syntax Directed Translation

Augment CFG rules with translation rules (at least 1 per production)

- Define translation of LHS nonterminal as function of
  - Constants
  - RHS nonterminal translations
  - RHS terminal value

Assign rules bottom-up
SDT Example

CFG

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B -&gt; 0</td>
<td>(B\text{.trans} = 0)</td>
</tr>
<tr>
<td>1</td>
<td>(B\text{.trans} = 1)</td>
</tr>
<tr>
<td>B 0</td>
<td>(B\text{.trans} = B_2\text{.trans} * 2)</td>
</tr>
<tr>
<td>B 1</td>
<td>(B\text{.trans} = B_2\text{.trans} * 2 + 1)</td>
</tr>
</tbody>
</table>

Input string

<table>
<thead>
<tr>
<th>Input</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10110</td>
<td>22</td>
</tr>
</tbody>
</table>

Translation is the value of the input
SDT Example 2: Declarations

CFG

\[
\begin{align*}
DList & \rightarrow \varepsilon \\
& \quad | \ DList \ Decl \\
Decl & \rightarrow Type \ id \\
Type & \rightarrow \text{int} \\
& \quad | \ \text{bool}
\end{align*}
\]

Translation is a String of ids

Translation:

Input string
int xx;
bool yy;

\[
\begin{align*}
DList & \rightarrow \varepsilon \\
& \quad | \ DList \ Decl \\
Decl & \rightarrow Type \ id \\
Type & \rightarrow \text{int} \\
& \quad | \ \text{bool}
\end{align*}
\]
Exercise Time

Only add declarations of type int to the output String.

Augment the previous grammar:

**CFG**

<table>
<thead>
<tr>
<th>Nonterminal</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DList$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
<tr>
<td>$Decl$</td>
<td>$Type$ $id$ $;$</td>
</tr>
<tr>
<td>$Type$</td>
<td>$int$</td>
</tr>
<tr>
<td></td>
<td>$</td>
</tr>
</tbody>
</table>

**Rules**

- $DList$.trans = ‘’
- $DList$.trans = $Decl$.trans + ‘ ‘ + $DList_2$.trans
- $Decl$.trans = $id$.value

Different nonterms can have different types

Rules can have conditionals
SDT Example 2b: ints only

CFG

\[
\begin{align*}
DList & \rightarrow \varepsilon \\
& \quad | \quad Decl \ DList \\
Decl & \rightarrow Type \ id \\
Type & \rightarrow int \\
& \quad | \quad bool
\end{align*}
\]

Rules

\[
\begin{align*}
DList\.trans &= "" \\
DList\.trans &= Decl\.trans + " " + DList2\.trans \\
\text{if (Type\.trans)} \{ Decl\.trans = id\.value \} \text{ else } \{ Decl\.trans = "" \} \\
Type\.trans &= true \\
Type\.trans &= false
\end{align*}
\]

Input string

int xx;
bool yy;

Different nonterms can have different types

Rules can have conditionals

Translation is a String of int ids only
SDT for Parsing

In the previous examples, the SDT process assigned different types to the translation:

- Example 1: tokenized stream to an integer value
- Example 2: tokenized stream to a (java) String

For parsing, we’ll go from tokens to an Abstract-Syntax Tree (AST)
Abstract Syntax Trees

- A condensed form of the parse tree
- Operators at internal nodes (not leaves)
- Chains of productions are collapsed
- Syntactic details omitted

Example: \((5+2)*8\)
Exercise #2

• Show the AST for:
  
  (1 + 2) * (3 + 4) * 5 + 6

 Expr  ->  Expr + Term
   |  Term
Term  ->  Term * Factor
   |  Factor
Factor -> intlit
   |  ( Expr )

Expr -> Expr + Term
Expr1.trans = MkPlusNode(Expr2.trans, Term.trans)
AST for Parsing

In previous slides we did our translation in two steps

– Structure the stream of tokens into a parse tree
– Use the parse tree to build an abstract syntax tree, throw away the parse tree

In practice, we will combine these into 1 step

**Question:** Why do we even need an AST?

– More of a “logical” view of the program
– Generally easier to work with
AST Implementation

How do we actually represent an AST in code?
ASTs in Code

Note that we’ve assumed a field-like structure in our SDT actions:

\[ DList\text{.}trans = Decl\text{.}trans + " " + DList_2\text{.}trans \]

In our parser, we’ll define classes for each type of nonterminal, and create a new nonterminal in each rule.

– In the above rule we might represent DList as

```java
public class DList{
    public String trans;
}
```

– For ASTs: when we execute an SDT rule
  • we construct a new node object for the RHS
  • propagate its fields with the fields of the LHS nodes
Thinking about implementing ASTs

Consider the AST for a simple language of Expressions

<table>
<thead>
<tr>
<th>Input</th>
<th>Tokenization</th>
<th>AST</th>
<th>Naïve AST Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 + 2</td>
<td>intlit plus intlit</td>
<td>+ 1 2</td>
<td>class PlusNode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IntNode left;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>IntNode right;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>class IntNode{</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>int value;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
Thinking about implementing ASTs

Consider AST node classes

- We’d like the classes to have a common inheritance tree

Naïve AST Implementation

```java
class PlusNode {
    IntNode left;
    IntNode right;
}

class IntNode {
    int value;
}
```

Naïve java AST

```
PlusNode
    IntNode left: 1
    IntNode right: 2

IntNode
    int value: 1

IntNode
    int value: 2
```
Thinking about implementing ASTs

Consider AST node classes
  – We’d like the classes to have a common inheritance tree

Naïve AST Implementation

```java
class PlusNode {
    IntNode left;
    IntNode right;
}
class IntNode {
    int value;
}
```

Better java AST

Make these extend ExpNode
Implementing ASTs for Expressions

CFG
Expr  ->  Expr + Term
  |  Term
Term  ->  Term * Factor
  |  Factor
Factor ->  intlit
  |  ( Expr )

Translation Rules
Expr1.trans = new PlusNode(Expr2.trans, Term.trans)
Expr.trans = Term.trans
Term1.trans = new TimesNode(Term2.trans, Factor.trans)
Term.trans = Factor.trans
Factor.trans = new IntNode(1)
Factor.trans = Expr.trans

Example: 1 + 2

Diagram of Expr node with children Term and Factor, and intlit nodes.
void foo(int x, int y) {
    if (x == y) {
        return;
    }
    while (x < y) {
        cout << "hello";
        x = x + 1;
    }
}

An AST for a code snippet
Summary (1 of 2)

Today we learned about

– Syntax-Directed Translation (SDT)
  • Consumes a parse tree with actions
  • Actions yield some result

– Abstract Syntax Trees (ASTs)
  • The result of SDT for parsing in a compiler
  • Some practical examples of ASTs
Summary (2 of 2)

Scanner

Language abstraction: RegEx
Output: Token Stream
Tool: JLex
Implementation: DFA walking via table

Parser

Language abstraction: CFG
Output: AST by way of Parse Tree
Tool: Java CUP
Implementation: ???

Next time

Next week