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

<table>
<thead>
<tr>
<th>CFG</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>B → 0</td>
<td>(B.\text{trans} = 0)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input string

10110

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

CFG | Rules
--- | ---
\[ DList \rightarrow \varepsilon \] | \[ DList_{trans} = "" \]
\[ DList \rightarrow DList\ Decl \] | \[ DList_{trans} = Decl_{trans} + " " + DList_{2}\_trans \]
\[ Decl \rightarrow Type\ id ; \] | \[ Decl_{trans} = id_{value} \]
\[ Type \rightarrow int \] | \[ \]
\[ Type \rightarrow bool \]

Input string
int xx;
bool yy;

Translation is a String of ids
Exercise Time

Only add declarations of type int to the output String.

Augment the previous grammar:

<table>
<thead>
<tr>
<th>CFG</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>DList</td>
<td>ε</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Decl DList</td>
</tr>
<tr>
<td>Decl</td>
<td>Type id ;</td>
</tr>
<tr>
<td>Type</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>bool</td>
</tr>
</tbody>
</table>

Different nonterms can have different types

Rules can have conditionals
SDT Example 2b: ints only

CFG Rules

\[\text{DList} \rightarrow \varepsilon \quad \text{DList}\.trans = "" \]

\[\mid \text{Decl} \text{DList} \quad \text{DList}\.trans = \text{Decl}\.trans + " " + \text{DList}2\.trans\]

\[\text{Decl} \rightarrow \text{Type id} ; \]

\[\text{Type} \rightarrow \text{int} \quad \text{Type}\.trans = \text{true}\]

\[\mid \text{bool} \quad \text{Type}\.trans = \text{false}\]

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 )
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?

We’ll take inspiration from how we represented tokens in JLex
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></td>
<td></td>
</tr>
</tbody>
</table>

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

class IntNode{
    int value;
}
```
Thinking about implementing ASTs

Consider AST node classes

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

```java
Naïve AST Implementation

class PlusNode
{
    IntNode left;
    IntNode right;
}

class IntNode
{
    int value;
}
```

Naïve java AST

AST

```
+ 1 2
```

PlusNode
IntNode left: IntNode right:
IntNode left: IntNode right:
IntNode value: 1
IntNode value: 2
Thinking about implementing ASTs

Consider AST node classes

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

```java
Naïve AST Implementation

class PlusNode {
    IntNode left;
    IntNode right;
}

class IntNode {
    int value;
}

Better java AST

Make these extend ExpNode

PlusNode
    ExpNode left: 1
    ExpNode right: 2

IntNode
    int value: 1
    int value: 2
```
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(intlit.value)
- Factor.trans = Expr.trans

Example: 1 + 2
An AST for an Harambe snippet

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

Next week