CS536

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 a function of
    • Constants
    • RHS nonterminal translations
    • RHS terminal value

• Assign rules bottom up
SDT Example

CFG

B -> 0
| 1
| B 0
| B 1

Rules

B.trans = 0
B.trans = 1
B.trans = B_2.trans * 2
B.trans = B_2.trans * 2 + 1

Input string

10110

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

CFG

\[
\begin{align*}
DList & \rightarrow \varepsilon \\
& \mid DList \ Decl \\
Decl & \rightarrow Type \ id \;
\end{align*}
\]

\[
\begin{align*}
Type & \rightarrow int \\
& \mid bool
\end{align*}
\]

Rules

\[
\begin{align*}
DList.\text{trans} & = "" \\
DList.\text{trans} & = Decl.\text{trans} + "" + DList_2.\text{trans} \\
Decl.\text{trans} & = \text{id}.\text{value}
\end{align*}
\]

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>$DList \rightarrow \epsilon$</td>
<td>$DList\text{.trans} = &quot;&quot;$</td>
</tr>
<tr>
<td>$\mid Decl\ DList$</td>
<td>$DList\text{.trans} = Decl\text{.trans} + &quot; &quot; + DList_2\text{.trans}$</td>
</tr>
<tr>
<td>$Decl \rightarrow Type\ id;$</td>
<td>$Decl\text{.trans} = id\text{.value}$</td>
</tr>
<tr>
<td>$Type \rightarrow \text{int}$</td>
<td></td>
</tr>
<tr>
<td>$\mid \text{bool}$</td>
<td></td>
</tr>
</tbody>
</table>

Different nonterms can have different types

Rules can have conditionals
SDT Example 2b: ints only

CFG

| DList → ε          |
| Decl DList         |

| Decl → Type id ;   |
| Type → int         |
| bool               |

Rules

\[
DList\.trans = ""
\]

\[
DList\.trans = Decl\.trans + " " + DList.\_trans
\]

if (type\.trans) \{Decl\.trans = id\.value\} else \{Decl\.trans = ""\}

Type\.trans = true
Type\.trans = false

Input string

int xx;
bool yy;

Different nonterms can have different types

Rules can have conditionals
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:
  
  \((1 + 2) \times (3 + 4) \times 5 + 6\)

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 define DList to be represented 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, and propagate its fields with the fields of the LHS nodes
Thinking about implementing ASTs

- Consider the AST for a simple language of Expressions

```
Input           Tokenization           AST
1 + 2           intlit plus intlit   +

Naïve AST Implementation

```java

class IntNode {
    int value;
}

class PlusNode {
    IntNode left;
    IntNode right;
}
```
Thinking about implementing ASTs

- Consider AST node classes
  - We’d like the classes to have a common inheritance tree
Thinking about implementing ASTs

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

```
Naïve AST Implementation

class PlusNode
{
    IntNode left;
    IntNode right;
}

class IntNode
{
    int value;
}
```

Better java AST

```
PlusNode
ExpNode left: 
ExpNode right: 
```

```
IntNode
int value: 1
```

```
IntNode
int value: 2
```

Make these extend ExpNode
### Implementing ASTs for Expressions

<table>
<thead>
<tr>
<th>CFG</th>
<th>Translation Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr -&gt; Expr + Term</td>
<td>Expr1.trans = new PlusNode(Expr2.trans, Term.trans)</td>
</tr>
<tr>
<td></td>
<td>Expr.trans = Term.trans</td>
</tr>
<tr>
<td>Term -&gt; Term * Factor</td>
<td>Term1.trans = new TimesNode(Term2.trans, Factor.trans)</td>
</tr>
<tr>
<td></td>
<td>Term.trans = Factor.trans</td>
</tr>
<tr>
<td>Factor -&gt; intlit</td>
<td>Factor.trans = new IntNode(intlit.value)</td>
</tr>
<tr>
<td></td>
<td>Factor.trans = Expr.trans</td>
</tr>
</tbody>
</table>

Example: $1 + 2$

![AST Diagram](image)
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
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