Announcements

Working in pairs is only allowed for programming assignments and not for homework problems
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 \( w \), and end with yes/no depending on whether \( w \in L(G) \)

CFGs in a compiler
- Start with a string \( w \), and end with a parse tree for \( w \) if \( w \in L(G) \)

Generally an abstract-syntax tree rather than a parse tree
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 (SDT)

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 -&gt; 0</td>
<td>B.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**

<table>
<thead>
<tr>
<th>Non-terminal</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>DList</td>
<td>$\epsilon$</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**

- $DList\.trans = ""$
- $DList\.trans = Decl\.trans + " " + DList_2\.trans$
- $Decl\.trans = id\.value$

Input string:

```
int xx;
bool yy;
```
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 → ε</td>
<td>DList.trans = “”</td>
</tr>
<tr>
<td></td>
<td>DList.trans = Decl.trans + “ “ + DList₂.trans</td>
</tr>
<tr>
<td>Decl DList</td>
<td>Decl.trans = id.value</td>
</tr>
<tr>
<td>Decl → Type id ;</td>
<td></td>
</tr>
<tr>
<td>Type → int</td>
<td>Rules can have conditionals</td>
</tr>
<tr>
<td></td>
<td>Different nonterms can have different types</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SDT Example 2b: ints only

CFG

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

Rules

\[
\begin{align*}
DList.trans & = "" \\
DList.trans & = Decl.trans + "" + DList_2.trans \\
Decl.trans & = (Type.trans \ ? Id.value : "") \\
Type.trans & = true \\
Type.trans & = false
\end{align*}
\]

Input string

int xx;
bool yy;

Different nonterms can have different types

Rules can use conditional expressions

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: \((5+2)\times8\)
Exercise #2

- Show the AST for:
  \[(1 + 2) \times (3 + 4) \times 5 + 6\]

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Expr} + \text{Term} \\
& \quad \mid \text{Term} \\
\text{Term} & \rightarrow \text{Term} \times \text{Factor} \\
& \quad \mid \text{Factor} \\
\text{Factor} & \rightarrow \text{intlit} \\
& \quad \mid (\text{Expr})
\end{align*}
\]

Expr \rightarrow Expr + Term
\quad \mid Term
Term \rightarrow Term \times Factor
\quad \mid Factor
Factor \rightarrow intlit
\quad \mid (Expr)

Expr \rightarrow Expr + Term
\quad \mid Term
Expr1.trans = MkPlusNode(Expr2.trans, Term.trans)
AST for Parsing

In previous slides we did the translation in two steps

- Structure the stream of tokens into a parse tree
- Use the parse tree to build an abstract-syntact tree; then throw away the parse tree

In practice, we will combine these into one step

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

- More of a “logical” view of the program: the essential structure
- Generally easier to work with an AST (in the later phases of name analysis and type checking)
  - no cascades of $\text{exp} \rightarrow \text{term} \rightarrow \text{factor} \rightarrow \text{intlit}$, which was introduced to capture precedence and associativity
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:

```
Expr -> Expr + Term  \textit{Expr1}.trans = \text{MkPlusNode}(\textit{Expr2}.trans, \textit{Term}.trans)
```

In our parser, we’ll define a class for each kind of ADT node, and create a new node object in some rules

- In the above rule we would represent the \textit{Expr1}.trans value via the class

  ```
  \text{public class PlusNode extends ExpNode \{}
  \text{  public ExpNode left;}
  \text{  public ExpNode right;}
  \text{\}}
  ```

- For ASTs: when we execute an SDT rule
  - we construct a new node object, which becomes the value of LHS.trans
  - populate the node’s fields with the translations of the RHS nonterminals
How to implement ASTs

Consider the AST for a simple language of Expressions

Input
1 + 2

Tokenization
intlit plus intlit

AST

Naïve AST Implementation

class PlusNode
    IntNode left;
    IntNode right;
}

class IntNode{
    int value;
}
How to implement 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: [value: 1]
IntNode right: [value: 2]

IntNode
int value: 1

IntNode
int value: 2
```
How to implement 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
- Make these fields be of class ExpNode

[Diagram showing the structure of AST nodes and class structures]
Implementing ASTs for Expressions

CFG

<table>
<thead>
<tr>
<th>Expr</th>
<th>-&gt;</th>
<th>Expr + Term</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Term</td>
</tr>
<tr>
<td>Term</td>
<td>-&gt;</td>
<td>Term * Factor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
</tr>
<tr>
<td>Factor</td>
<td>-&gt;</td>
<td>intlit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( Expr )</td>
</tr>
</tbody>
</table>

Example: 1 + 2

Translation Rules

- $Expr_1.trans = \text{new PlusNode}(Expr_2.trans, Term.trans)$
- $Expr.trans = \text{Term.trans}$
- $Term_1.trans = \text{new TimesNode}(Term_2.trans, Factor.trans)$
- $Term.trans = Factor.trans$
- $Factor.trans = \text{new IntNode}(\text{intlit}.value)$
- $Factor.trans = Expr.trans$
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 an SDT performed during parsing in a compiler
  • Some practical examples of ASTs
Summary (2 of 2)

Scanner
Language abstraction: RegExp
Output: Token Stream
Tool: JLex
Implementation: Interpret DFA using table (for \( \delta \)), recording
  most_recent_accepted_position and most_recent_token

Parser
Language abstraction: CFG
Output: AST by way of a syntax-directed translation
Tool: Java CUP
Implementation: ???

Next week

Next week