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**

B -> 0 \( B \).\text{trans} = 0
| 1 \( B \).\text{trans} = 1
| \( B\ 0 \) \( B \).\text{trans} = \( B_2 \).\text{trans} * 2
| 2 \( B\ 1 \) \( B \).\text{trans} = \( B_2 \).\text{trans} * 2 + 1

**Rules**

**Input string**

10110

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

CFG

\[ DList \rightarrow \varepsilon \]

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

Rules

\[ DLList \text{.trans} = "" \]

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

\[ Type \rightarrow \text{int} \]

\[ | \text{bool} \]

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:

```
CFG       Rules
DList     → e      DList.trans = ""
           | DList Decl     DList.trans = Decl.trans + " " + DList₂.trans
Decl     → Type id ;       Decl.trans = id.value
Type     → int
           | bool
```

Different nonterms can have different types.
Rules can have conditionals.
DT Example 2b: ints only

CFG

\[ DList \rightarrow \epsilon \]

\[ \quad \mid Decl \ DList \ DList.trans = Decl.trans + " " + DList_2.trans \]

Rules

\[ DList.trans = "" \]

\[ Decl \ DList \ DList.trans = Decl.trans + " " + Decl.trans = id.value \]

Type \rightarrow \text{int}

\[\text{Input string } \ \text{bool} \]

\[ \text{int xx;} \]

\[ \text{bool yy;} \]

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:
  \[(1 + 2) \times (3 + 4) \times 5 + 6\]

```plaintext
 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>
</tr>
</thead>
<tbody>
<tr>
<td>1 + 2</td>
<td>intlit plus intlit</td>
<td>+ 1 2</td>
</tr>
</tbody>
</table>

Naïve AST Implementation

```java
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

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

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

IntNode
int value: 1
```

Make these extend ExpNode
Implementing ASTs for Expressions

CFG

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

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

```
不影响
```

```
Expr
  +
  Term
    *
    Factor
      intlit
        intlit 1
  intlit
    intlit 2

PlusNode
ExpNode left: 1
ExpNode right: 2
```

```
IntNode
value: 1
```

```
IntNode
value: 2
```
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 time

Next week