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 a 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 $w \in L(G)$
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

#### CFG

<table>
<thead>
<tr>
<th>Rule</th>
<th>SDT</th>
</tr>
</thead>
<tbody>
<tr>
<td>B -&gt; 0</td>
<td>B.trans = 0</td>
</tr>
<tr>
<td>1</td>
<td>B.trans = 1</td>
</tr>
<tr>
<td>B 0</td>
<td>B.trans = B₂.trans * 2</td>
</tr>
<tr>
<td>B 1</td>
<td>B.trans = B₂.trans * 2 + 1</td>
</tr>
</tbody>
</table>

#### Rules

- B.trans = 0
- B.trans = 1
- B.trans = B₂.trans * 2
- B.trans = B₂.trans * 2 + 1

#### Input string

10110

#### Translation

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

CFG

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

Rules

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

Translation is a String of ids

Input string

int xx;
bool yy;

Diagram:

```
Translation is a String of ids
```

```
Input string
int xx;
bool yy;

```

```
Translation is a String of ids
```

```
Input string
int xx;
bool yy;

```
Exercise Time

Only add declarations of type int to the output String.

Augment the previous grammar:

### CFG

- **DList** → $\epsilon$
  - | **Decl** **DList**
- **Decl** → **Type** id ;
- **Type** → int
  - | **bool**

### Rules

- **DList**.trans = “”
- **DList**.trans = **DList**$_2$.trans + “ “ + **Decl**.trans
- **Decl**.trans = **id**.value

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

CFG

- **DList** → \( \varepsilon \)
- **Decl** DList
- **Type** → int
- **bool**

Rules

- **DList**.trans = “”
- **DList**.trans = **DList**.trans + “ “ + **Decl**.trans
- **Decl**.trans = (**Type**.trans ? **id**.value : “”)
- **Type**.trans = true
- **Type**.trans = false

Input string

- int xx;
- bool yy;

Different nonterms can have different types

Rules can use conditional expressions
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 )
```

Expr -> Expr + Term  

\[Expr1\.trans = \text{MkPlusNode}(Expr2\.trans, Term\.trans)\]
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-syntax 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 exp → term → factor → 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:

\[ \text{Expr} \rightarrow \text{Expr} + \text{Term} \quad \text{Expr1}.\text{trans} = \text{MkPlusNode} (\text{Expr2}.\text{trans}, \text{Term}.\text{trans}) \]

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 \text{Expr1}.\text{trans} value via the class

```java
public class PlusNode extends ExpNode {
    public ExpNode left;
    public ExpNode right;
}
```

- 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

Naïve AST Implementation

```java
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:
IntNode right:

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 fields be of class ExpNode

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(intlit.value)
Factor.trans = Expr.trans

Example: 1 + 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 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