CS 536 Announcements for Thursday, February 13, 2025

Programming Assignment 2 – due Tuesday, February 18

Homework 2 – is available

Last Time

- why regular expressions aren't enough
- CFGs
  - formal definition
  - examples
  - language defined by a CFG
- parse trees
- Makefiles
- ambiguous grammars
- grammars for expressions
  - precedence
  - associativity
- grammars for lists

Today

- syntax-directed translation
- abstract syntax trees
- implementing ASTs

CFG review

```
prog   → BEGIN stmts END

stmts  → stmts SEMICOLON stmt
       |  stmt
stmt   → ID ASSIGN expr

expr   → expr PLUS term
       |  term

term   → term TIMES factor
       |  factor

factor → expon POW factor
       |  expon

expon  → INTLIT
       |  LPAREN expr RPAREN
```

Overview of CFGs

CFGs for language definition

- the CFGs we've discussed can generate/define languages of valid strings

CFGs for language recognition

CFGs for parsing

Syntax-directed translation

= translating from a sequence of tokens into a sequence of actions/other form, based on underlying syntax

To define a syntax-directed translation

Augment CFG with ***translation rules***

- define translation of LHS non-terminal as a function of
  -
  -
  -

To translate a sequence of tokens using SDT

-
- use translation rules to compute translation of
- translation of sequence of tokens is

The **type** of the translation can be anything:

Note:

Example: grammar for language of binary numbers

CFG	translation rules
b → 0	b.trans = 0
| 1	b.trans = 1
| b 0	b_1.trans = b_2.trans * 2
| b 1	b_1.trans = b_2.trans * 2 + 1

Example: grammar for language of variable declarations

CFG			Translation rules
declList	→	ε	
	|	decl declList	
decl	→	type ID **;**	
type	→	INT	
	|	BOOL	

Write a syntax-directed translation for the CFG given above so that the translation of a sequence of tokens is a string containing the ID's that have been declared.

Example: grammar for language of variable declarations

<u>CFG</u>			<u>Translation rules</u>
declList	→	ε	
	|	decl declList	
decl	→	type ID **;**	
type	→	INT	
	|	BOOL	

Modify the previous syntax-directed translation so that only declarations of type `int` are added to the output string.

SDT for parsing

Previous examples showed SDT process assigning different types to the translation

- translate tokenized stream to an integer value
- translate tokenized stream to a string

For parsing, we'll need to translate a tokenized stream to an **abstract-syntax tree (AST)**

Abstract syntax trees

AST = condensed form of parse tree

-
-
-
-

AST Example

<u>CFG</u>

```
expr   →  expr PLUS term
       |  term
term   →  term TIMES factor
       |  factor
factor →  INTLIT
       |  LPAREN expr RPAREN
```

SDT review

SDT = translating from a sequence of tokens into a sequence of actions/other form, based on underlying syntax

To define a syntax-directed translation

- augment CFG with ***translation rules***
  - define translation of LHS non-terminal as a function of:
    - constants
    - translations of RHS non-terminals
    - values of terminals (tokens) on RHS

To translate a sequence of tokens using SDT (conceptually)

- build parse tree
- use translation rules to compute translation of each non-terminal (bottom-up)
- translation of sequence of tokens = translation of parse tree's root non-terminal

For parsing, we'll need to translate tokenized stream to **abstract-syntax tree (AST)**

Example

```
expr   → expr + term
       |  term
term   → term * factor
       |  factor
factor → INTLIT
       |  ( expr )
```

AST for parsing

We've been showing the translation in two steps:

In practice we'll do

Why have an AST?

AST implementation

Define a class for each kind of AST node

Create a new node object in some rules

- new node object is the value of LHS.trans
- fields of node object come from translations of RHS non-terminals

Translation rules to build ASTs for expressions

CFG			Translation rules
expr	→	expr + term	$expr_1$.trans =
	|	term	expr.trans =
term	→	term * factor	$term_1$.trans =
	|	factor	term.trans =
factor	→	INTLIT	factor.trans =
	|	(expr)	factor.trans =

ASTs for non-expressions

Example

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

ASTs for lists

CFG

idList → idList COMMA ID
| ID

The bigger picture

Scanner

- **Language abstraction**: regular expressions
- **Output**: token stream
- **Tool**: JLex
- **Implementation**: interpret DFA using table (for δ),
  recording most_recent_accepted_position & most_recent_token

Parser

- **Language abstraction**:
- **Output**:
- **Tool**:
- **Implementation**:

Next Time

- Java CUP
- approaches to parsing
- bottom-up parsing
- CFG transformations
- CYK algorithm