The Structure of Programming Languages

- With the exception of the Generator we saw that all language processors perform some kind of syntax analysis – an analysis of the structure of the program.
- To make this efficient and effective we need some mechanism to specify the structure of a programming language in a straightforward manner.
- We use grammars for this purpose.
Grammars

- The most convenient way to describe the structure of programming languages is using a context-free grammar (often called CFG or BNF for Backus-Nauer Form).
- Here we will simply refer to grammars with the understanding that we are referring to CFGs. (there are many kind of other grammars: regular grammars, context-sensitive grammars, etc)
Grammars

Grammars can readily express the structure of phrases in programming languages

- stmt: function-def | return-stmt | if-stmt | while-stmt
- function-def: function name expr stmt
- return-stmt: return expr
- if-stmt: if expr then stmt else stmt endif
- while-stmt: while expr do stmt enddo
Grammars

- Grammars have 4 parts to them
  1. Non-terminal Symbols - these give names to phrase structures - e.g. function-def
  2. Terminal Symbols - these give names to the tokens in a language – e.g. while (sometimes we don’t use explicit tokens but put the words that make up the tokens of a language in quotes)
  3. Rules - these describe that actual structure of phrases in a language – e.g. return-stmt: return exp
  4. Start Symbol - a special non-terminal that gives a name to the largest possible phrase(s) in the language (often denoted by an asterisk)
    - In our case that would probably be the stmt non-terminal
Example: The Exp0 Language

```plaintext
grammar exp0;

prog   : stmt+ ;

stmt   : 'p' exp ';' |
       | 's' lhsvar exp ';' ;

exp    : '+' exp exp |
       | '-' exp exp |
       | '(' exp ')' |
       | rhsvar |
       | num ;

lhsvar : 'x' | 'y' | 'z' ;

rhsvar : 'x' | 'y' | 'z' ;

num    : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
```

Start Symbol: prog

More than one statement allowed
Grammars

- A grammar tells us if a sentence belongs to the language,
  - e.g. Does ‘s x 3 ;’ belong to the language?
- We can show that a sentence belongs to the language by constructing a parse tree starting at the start symbol.
Grammars

s x 3;

grammar exp0;

prog : stmt+

stmt : 'p' exp ';'
      | 's' lhsvar exp ';'

exp : '+’ exp exp
     | '-' exp exp
     | '(' exp ')' rhsvar
     | num

lhsvar : 'x' | 'y' | 'z'

rhsvar : 'x' | 'y' | 'z'

num : '0' ... '9'

Note: constructing the parse tree by filling in the leftmost non-terminal at each step we obtain the left-most derivation:

prog ➝ stmt ➝ s lhsvar exp ; ➝ s x exp ; ➝ s x 3 ;

Constructing the parse tree by filling in the rightmost non-terminal at each step we obtain the right-most derivation.
Grammars

- Every valid sentence (a sentence that belongs to the language) has a parse tree.
- Test if these sentences are valid:
  - \( p \ x + 1 \); 
  - \( s \ x 1 ; s \ y \ x \);
  - \( s \ x 1 ; p (+ x 1) \);
  - \( s \ y + 3 \ x \);
  - \( s + y 3 \ x \);
Parsers

- The converse is also true!
- If a sentence has a parse tree, then it belongs to the language.
- This is precisely what parsers do: to show a program is syntactically correct parsers construct a parse tree
Top-Down Parsers - LL(1)

- LL(1) parsers start constructing the parse tree at the start symbol
  - as opposed to bottom up parsers, LR(1)
- LL(1) parsers use the current position in the input stream and a single look-ahead token decide how to construct the next node(s) in the parse tree.
- LL(1)
  - Reads input from Left to right.
  - Constructs the Leftmost derivation
  - Uses 1 look-ahead token.
Consider: p + x 1 ;

For top-down parsing we can think of the grammar extended with the one token look-ahead set.

The look-ahead set uniquely identifies the selection of each rule within a block of rules.
function ComputeLookaheadSets(G) returns G'
// G is a context-free grammar viewed as a list of rules
// G' is a context-free grammar extended with lookahead sets
begin
    let G' be an empty list of rules
    for each rule A → RuleBody in G do
        let S be the first symbol in RuleBody
        if S is the empty string then
            append the rule A → {""} "" to G'
        else if S is a terminal symbol then
            append the rule A → {S} RuleBody to G'
        else if S is a non-terminal symbol then
            let L := LookaheadSet(S, G)
            append the rule A → L RuleBody to G'
        end if
    end do
    return G'
end
Computing the Lookahead Set

```python
function LookaheadSet(N, G) returns L
    // N is a non-terminal in G
    // G is a context-free grammar
    // L is a lookahead set
    begin
        let L be the empty set
        for each rule N → RuleBody in G do
            let Q be the first symbol in RuleBody
            if Q is the empty string then
                report an error and abort
            else if Q is a terminal symbol then
                let L := L ∪ {Q}
            else if Q is a non-terminal symbol then
                let L := L ∪ LookaheadSet(Q, G)
            end if
        end do
        return L
    end
```
Computing the Lookahead Set

**grammar G:**

```plaintext
prog : stmt+
    ;
stmt : 'p' exp ';'
    | 's' lhsvar exp ';
    ;
exp : '+' exp exp
    | '-' exp exp
    | '(' exp ')' 
    | rhsvar 
    | num 
    ;
lhsvar : 'x' | 'y' | 'z'
    ;
rhsvar : 'x' | 'y' | 'z'
    ;
num : '0' | '1' | ... | '9'
    ;
```

**grammar G':**

```plaintext
prog : {'p','s'} stmt+
    ;
stmt : { 'p' } 'p' exp ';
    | { 's' } 's' lhsvar exp ';
    ;
exp : { '+' } '+' exp exp
    | { '-' } '-' exp exp
    | { '(' } '(' exp ')' 
    | { 'x', 'y', 'z' } rhsvar 
    | { '0','1',...,'9' } num 
    ;
lhsvar : { 'x' } 'x' | { 'y' } 'y' | { 'z' } 'z'
    ;
rhsvar : { 'x' } 'x' | { 'y' } 'y' | { 'z' } 'z'
    ;
num : { '0' } '0' | { '1' } '1' | ... | { '9' } '9'
    ;
```
Computing the Lookahead Set

- Actually, the algorithm we have outlined computes the lookahead set for a simpler parsing technique called sLL(1) – simplified LL (1) parsing.

- Full LL(1) parsing has to deal with non-terminals that expand into the empty string in the first position of a production.

- All our hand-built parsers will be sLL(1) but when we use ANTLR we will have access to full LL(1) parsing.
Constructing a Parser

- A sLL(1) parser can be constructed by hand by converting each non-terminal into a function.
- The body of the function implements the right sides of the rules for each non-terminal:
  - Process terminals
  - Call the functions of other non-terminals as appropriate
Constructing a Parser

- We need two auxiliary functions and a driver:

```java
function char peekToken() { return the next non-whitespace character in the input stream without removing the character from the input stream. Ignores all whitespace characters. }

function void matchInput(char c) throws SyntaxErrorException { match the character ‘c’ against the current character in the input stream and remove the character ‘c’ from the input stream. Throws an exception if the character is not found. Ignores all whitespace characters. }
```

```java
function main() {
    prog();
}
```

Start symbol
Constructing a Parser

Consider the following rule:

\[
\text{prog} \quad : \quad \text{stmt}^+ 
\]

function prog () {
    do {
        stmt();
    } while(peekToken() != EOF);
}

Note: a lookahead set is not necessary here – only one rule to choose from.
Constructing a Parser

```plaintext
stmt : {'p'} 'p' exp ';'
| {'s'} 's' lhsvar exp ';
```

```javascript
function stmt () {
    switch(peekToken()) {
        case 'p':
            matchInput('p');
            exp();
            matchInput(';');
            break;
        case 's':
            matchInput('s');
            lhsvar();
            exp();
            matchInput(';;');
            break;
        default:
            throw(new SyntaxErrorException);
    }
}
```

Notice that we are using the look-ahead set to decide which rule to call!
function exp () {
    switch(peekToken()) {
        case '+':
            matchInput('+');
            exp();
            exp();
            break;
        case '-':
            matchInput('-');
            exp();
            exp();
            break;
        case '(':
            matchInput('(');
            exp();
            matchInput(')');
            break;
        case 'x':
        case 'y':
        case 'z':
            rhsvar();
            break;
        case '0':
        case '1':
            
            break;
        case '9':
            num();
            break;
        default:
            throw(new SyntaxErrorExcption());
    }
}
Constructing a Parser

```javascript
function lhsvar () {
  // match the possible variable names
  switch(peekToken()) {
    case 'x':
      matchInput('x');
      break;
    case 'y':
      matchInput('y');
      break;
    case 'z':
      matchInput('z');
      break;
    default:
      throw(new SyntaxErrorException());
  }
}
```

lhsvar : { 'x' } 'x' | { 'y' } 'y' | { 'z' } 'z'
Constructing a Parser

```
function rhsvar () {
    // match the possible variable names
    switch(peekToken()) {
        case 'x':
            matchInput('x');
            break;
        case 'y':
            matchInput('y');
            break;
        case 'z':
            matchInput('z');
            break;
        default:
            throw(new SyntaxErrorException());
    }
}
```
Constructing a Parser

```
num : { '0' }'0' | { '1' }'1' | ... | { '9' }'9'
```

```javascript
function num () {
    // match the possible numbers
    switch(peekToken()) {
        case '0':
            matchInput('0');
            break
        case '1':
            matchInput('1');
            break;
        .
        .
        .
        case '9':
            matchInput('9');
            break;
        default:
            throw(new SyntaxErrorException());
    }
}
```
Constructing a Parser: An Example

```javascript
function prog () {
  do {
    stmt();
  } while(peekToken() != EOF);
}

function stmt () {
  switch(peekToken()) {
    case 'p':
      matchInput('p');
      exp();
      matchInput(';');
      break;
    case 's':
      matchInput('s');
      lhsvar();
      exp();
      matchInput(';');
      break;
    default:
      throw(new SyntaxErrorException());
  }
}

function exp () {
  switch(peekToken()) {
    case '+':
      matchInput('+');
      exp();
      exp();
      break;
    case '-':
      matchInput('-');
      exp();
      exp();
      break;
    case '(':
      matchInput('(');
      exp();
      matchInput(')');
      break;
    case 'x':
    case 'y':
    case 'z':
      rhsvar();
      break;
    case '0':
    case '1':
    case '9':
      num();
      break;
    default:
      throw(new SyntaxErrorException());
  }
}
```

Call Tree:

```
p + x 1 ;
```

```
prog() stmt() exp() exp() rhsvar() exp() num() matchInput('1')
```
Constructing a Parser: An Example

Observations:

- Our parser is an LL(1) parser (why?)
- The parse tree is implicit in the function call activation record stack
- Building a parser by hand is a lot of work and the parser is difficult to maintain.
- We would like a tool that reads our grammar file and converts it automatically into a parser – that is what ANTLR does!
Assignments

- Read Chapter 3 – reference guide
- install UbuntuBox from the course website
- Programming Assignment #1 -- see the website