Undergraduate Compilers Review

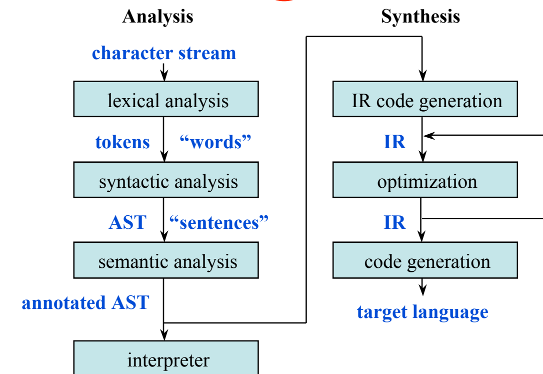
Announcements

- Send me email if you have set up a group. Include account names for group.
- Each project is 10% of grade

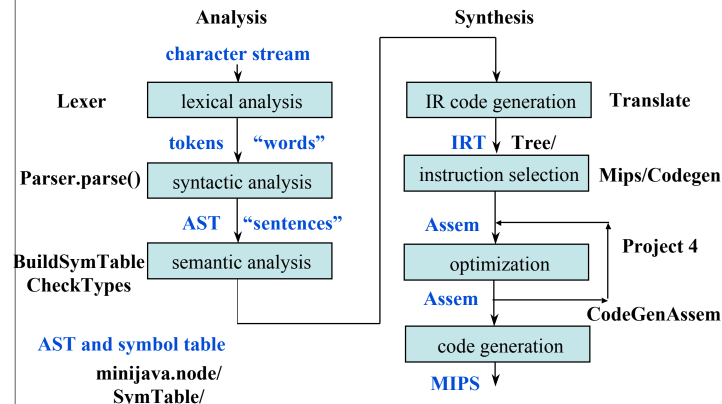
Today

- Outline of planned topics for course
- Overall structure of a compiler
- Lexical analysis (scanning)
- Syntactic analysis (parsing)

Structure of a Typical Interpreter Compiler



Structure of the MiniJava Compiler (CodeGenAssem.java)



Lexical Analysis (Scanning)

Break character stream into tokens ("words")

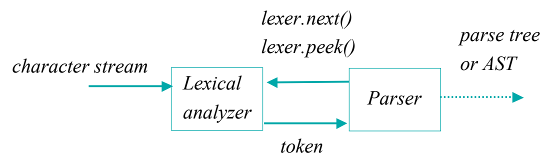
- Tokens, lexemes, and patterns
- Lexical analyzers are usually automatically generated from patterns (regular expressions) (*e.g.,* lex)

Examples

token	lexeme(s)	pattern
const	`const`	`const`
if	`if`	`if`
relation	`<,<=,=,!=,...`	`< | <= | = | != | ...`
identifier	`foo,index`	`[a-zA-Z_]+[a-zA-Z0-9_]*`
number	`3.14159,570`	`[0-9]+ | [0-9]*.[0-9]+`
string	`"hi", "mom"`	`".*"`

`const pi := 3.14159` ⇒ *const, identifier*(`pi`), *assign,number*(**3.14159**)

Interaction Between Scanning and Parsing



Specifying Tokens with SableCC

Theory meets practice:
- Regular expressions, formal languages, grammars, parsing…

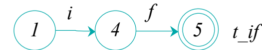
SableCC example input file:

```
Package minijava;

Helpers
  all = [0..0xFFFF];
  cr  = 13;

  digit  = ['0'..'9'];
  letter = ['a'..'z'] | ['A'..'Z'];
  underscore  = '_';

  not_star       = [all - '*'];
  not_star_slash = [not_star - '/'];

  c_comment = '/*' not_star* ('*'
(not_star_slash not_star*)?)* '*/';
```

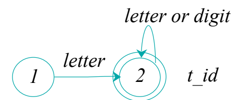
```
Tokens
 t_plus  = '+';

 t_if    = 'if';

 t_id = letter (letter | digit | underscore)*;

 t_blank   = (' ' | eol | tab)+;
 t_comment = c_comment | line_comment;

Ignored Tokens
  t_blank,
  t_comment;
```


Recognizing Tokens with DFAs

`'if'`



`letter (letter | digit)*`



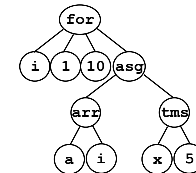
Ambiguity due to matching substrings
- Longest match
- Rule priority

Syntactic Analysis (Parsing)

Impose structure on token stream
- Limited to syntactic structure (⇒ high-level)
- Structure usually represented with an *abstract syntax tree* (AST)
- Parsers are usually automatically generated from context-free grammars (*e.g.,* yacc, bison, cup, javacc, sablecc)

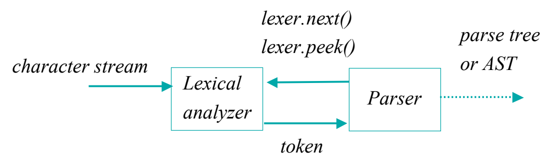
Example

```
for i = 1 to 10 do
   a[i] = x * 5;
```



for id(i) equal number(1) to number(10) do
id(a) lbracket id(i) rbracket equal id(x) times number(5) semi

Interaction Between Scanning and Parsing



Bottom-Up Parsing: Shift-Reduce

Grammer **a + b + c**

```
(1) S -> E
(2) E -> E + T
(3) E -> T
(4) T -> id
```

```
S -> E
  -> E + T
  -> E + id
  -> E + T + id
  -> E + id + id
  -> T + id + id
  -> id + id + id
```

Rightmost derivation: expand rightmost non-terminals first

SableCC, yacc, and bison generate shift-reduce parsers:

- LALR(1): look-ahead, left-to-right, rightmost derivation in reverse, 1 symbol lookahead
- LALR is a parsing table construction method, smaller tables than canonical LR

Reference: Barbara Ryder's 198:515 lecture notes

LR Parse Table

	Action			Goto		
State	**+**	**id**	**$**	**S**	**E**	**T**
0		s4			1	2
1	s3		accept			
2	r(3)	r(3)	r(3)			
3		s4				5
4	r(4)	r(4)	r(4)			
5	r(2)	r(2)	r(2)			

```
(1) S -> E
(2) E -> E + T
(3) E -> T
(4) T -> id
```

Look at current state and input symbol to get action
- shift(n): advance input, push n on stack
- reduce(k): pop rhs of grammar rule k, k = (lhs -> rhs)
  - look up state on top of stack and lhs for goto n
  - push n
- accept: stop and success
- error: stop and fail

Shift-Reduce Parsing Example

Stack	Input	Action
$ 0	a + b + c	shift 4
$ 0 a 4	+ b + c	reduce (4)
$ 0 T 2	+ b + c	reduce (3)
$ 0 E 1	+ b + c	shift
$ 0 E 1 + 3	b + c	shift
$ 0 E 1 + 3 b 4	+ c	reduce (4)
$ 0 E 1 + 3 T 5	+ c	reduce (2)
$ 0 E 1	+ c	shift
$ 0 E 1 + 3	c	shift
$ 0 E 1 + 3 c 4		reduce (4)
$ 0 E 1 + 3 T 5		reduce (2)
$ 0 E 1		accept

```
(1) S -> E
(2) E -> E + T
(3) E -> T
(4) T -> id
```

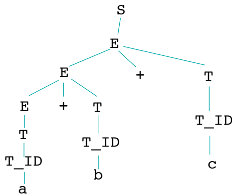
Reference: Barbara Ryder's 198:515 lecture notes

Syntax-directed Translation: AST Construction example

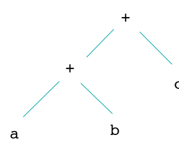
Grammer with production rules

```
S:  E        { $$ = $1; };
E:  E '+' T  { $$ = new node("+", $1, $3); }
   | T       { $$ = $1; }
   ;
T: T_ID      { $$ = new leaf("id", $1); };
```

Implicit parse tree for a+b+c



AST for a+b+c



Reference: Barbara Ryder's 198:515 lecture notes

Using SableCC to specify grammar and generate AST

```
Productions
cst_program {-> program} =
  cst_main_class cst_class_decl*
  {-> New program(cst_main_class.main_class,[cst_class_decl.class_decl])}
;

 cst_exp_list {-> exp* } =
  {many_rule}  cst_exp cst_exp_rest*
  {-> [cst_exp.exp, cst_exp_rest.exp] }

  |  {empty_rule}
     {-> □ }
  ;

 cst_exp_rest {-> exp* } = t_comma cst_exp
                            {-> [cst_exp.exp] };

 cst_type {-> type } = ...
     |    {int_rule} int
          {-> New type.int() }
```

```
Abstract Syntax Tree

  program =
    main_class [class_decls]:class_decl*;

  exp =
    {call}      exp t_id [args]:exp*  | ..

  type = {int} | {bool} | ...
```


AST, SymTable, IRT Tree, Assem, and MIPS for example

```
class And {
  public static void main(String[] a){
    System.out.println(new Foo().testing(42));    } }
class Foo {
  public int testing(int p) {
    int x;
    if (p < 10 && 2 < p) {
      x = 7;
    } else {
      x = 22;
    }
    return x;
}}
```


Concepts

Compilation stages in a compiler
- Scanning, parsing, semantic analysis, intermediate code generation, optimization, code generation

Lexical analysis or scanning
- Tools: SableCC, lex, flex, etc.

Syntactic analysis or parsing
- Tools: SableCC, yacc, bison, etc.

Parsing Terms
- see attached slides, be familiar with these terms

Next Time

Lecture

- More undergraduate compilers review

Parsing Terms

CFG (Context-free Grammer)

- production rule
- terminal
- nonterminal

BNF (Backus-Naur Form) and EBNF (Extended BNF): equivalent to CFGs

Parsing Terms cont …

Top-down parsing

- **LL(1):** left-to-right reading of tokens, leftmost derivation, 1 symbol look-ahead
- **Predictive parser**: an efficient non-backtracking top-down parser that can handle LL(1)
- More generally **recursive descent** parsing may involve backtracking

Bottom-up Parsing

- **LR(1):** left-to-right reading of tokens, rightmost derivation in reverse, 1 symbol lookahead
- **Shift-reduce parsers:** for example, bison, yacc, and SableCC generated parsers
- Methods for producing an LR parsing table
  - SLR, simple LR
  - Canonical LR, most powerful
  - **LALR(1)**