

Scanning and Parsing

Announcements

- Pick a partner by Monday
- Makeup lecture will be on Monday August 29th at 3pm

Today

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

Topics

I. The Basics

- Scanning and parsing
- Dataflow analysis
- Control flow analysis

II. Analyses and Representations

- SSA Form
- Redundancy elimination
- Aliases
- Interprocedural analysis

III. Low-Level Optimizations

- Register allocation
- Instruction scheduling
- Profile-guided and dynamic optimizations

Topics (cont)

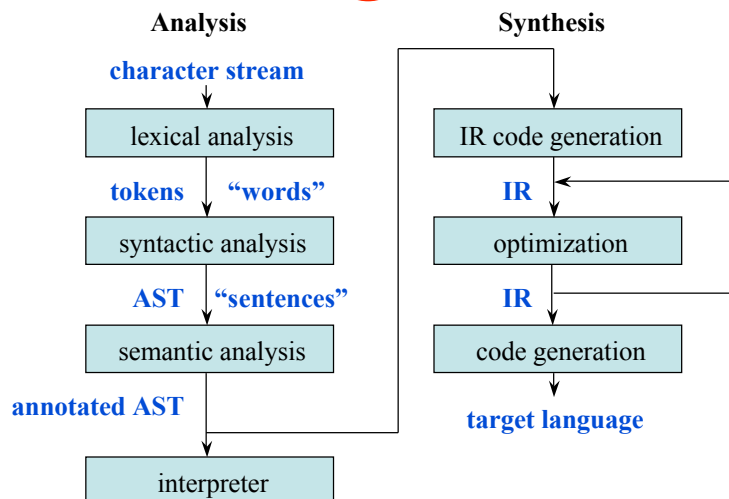
IV. High-Level Optimizations

- Dependence analysis
- Loop transformations
- Tiling
- Object-oriented optimizations

V. Emerging Topics

- Run-time reordering transformations
- Security and program checking
- Domain-specific program analysis and transformation

Structure of a Typical ~~Interpreter~~ Compiler



Lexical Analysis (Scanning)

Break character stream into tokens (“words”)

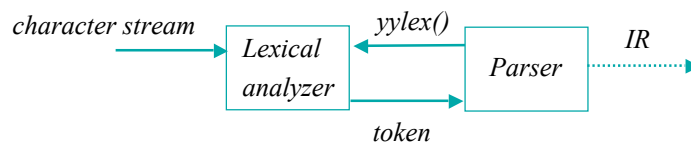
- Tokens, lexemes, and patterns
- Lexical analyzers are usually automatically generated from patterns (regular expressions) (e.g., lex)

Examples

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

const pi := 3.14159 \Rightarrow *const, identifier(pi), assign, number(3.14159)*

Interaction Between Scanning and Parsing



Specifying Tokens with Flex

Theory meets practice:

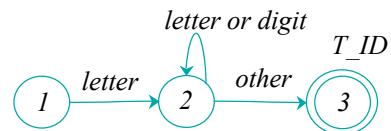
- Regular expressions, formal languages, grammars, parsing...

Flex example input file:

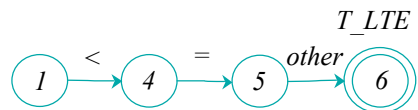
<pre>%{ #include <stdlib.h> #include "top-token.h" }% DIGIT [0-9] ID [a-zA-Z][a-zA-Z0-9]* %% [=\{\}] { return yytext[0]; } if { return T_IF; } => { return T_MAPSTO; }</pre>	<pre>"\\\/"[^\\n]*"\\n" // eat up one-line comments [\t\\n]+ // eat up whitespace {ID} { char *retstr; retstr = malloc(strlen(yytext)+1); strcpy(retstr,yytext); yyval.sval = retstr; return T_ID; }</pre>
---	---

Recognizing Tokens with DFAs

`[a-zA-Z][a-zA-Z0-9]*`



`<=`



`[=\{\}]`

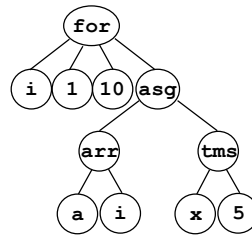
Syntactic Analysis (Parsing)

Impose structure on token stream

- Limited to syntactic structure (\Rightarrow high-level)
- Structure usually represented with an *abstract syntax tree* (AST)
- Parsers are usually automatically generated from grammars (e.g., yacc, bison, cup, javacc)

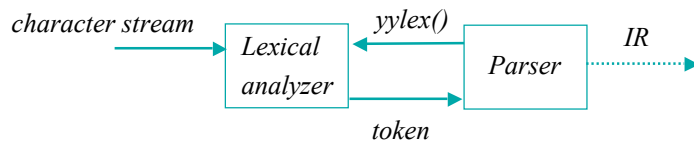
Example

```
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



Using bison or yacc with flex or lex

bison assumes that yylex() function has been defined.

bison example input file:

```
%union {
    char*   sval;
    int     ival;
    Expr*   exprptr;
    std::list<Stmt*> stmtlistptr;
};

%token <sval> T_STR_LITERAL
%token <ival> T_INT_LITERAL
%token T_IF T_THEN T_ELSE

%type <exprptr> Expr
%type <stmtlistptr> StmtList

Proc:      StmtList ;
StmtList:  StmtList Stmt | /*empty*/ ;
Stmt:     T_IF Expr T_THEN StmtList T_ELSE StmtList | /*other stmts*/ ;
```

Shift-Reduce Parsing

Parsing Terms

CFG (Context-free Grammar)

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

Parsing Terms cont ...

Top-down parsing

- **LL(1)**: left-to-right reading of tokens, leftmost derivation, 1 symbol lookahead
- **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 and yacc generated parsers
- Methods for producing an LR parsing table
 - SLR, simple LR
 - Canonical LR, most powerful
 - **LALR(1)**

Project 1: Scanners and Parsers for OpenAnalysis Test Input

```
// int main() {
    PROCEDURE = { < ProcHandle("main"), SymHandle("main") > }

    // int x;
    LOCATION = { < SymHandle("x"), local > }
    // int *p;
    LOCATION = { < SymHandle("p"), local > }

    // all other symbols visible to this procedure
    LOCATION = { < SymHandle("g"), not local > }

    // x = g;
    MEMREFEXPRS = { StmtHandle("x = g;") =>
        [
            MemRefHandle("x_1") => < SymHandle("x"), DEF >
            MemRefHandle("g_1") => < SymHandle("g"), USE >
        ] }
}
```



OpenAnalysis



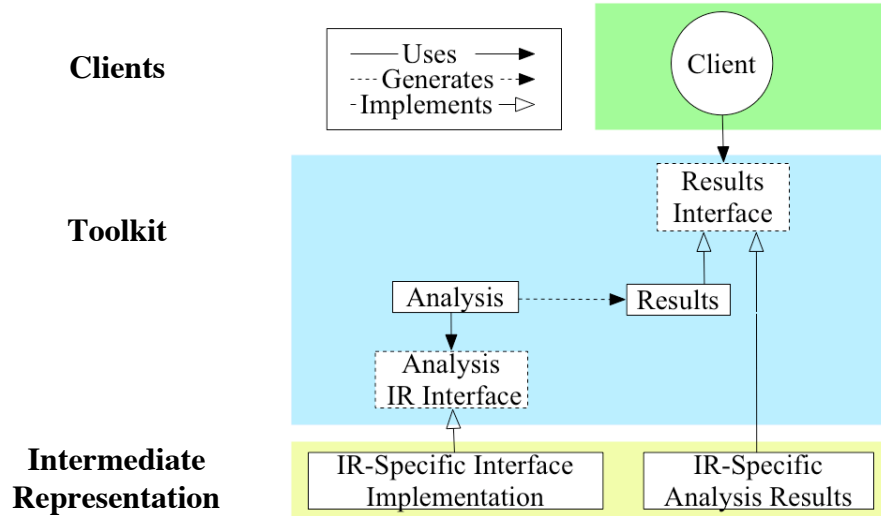
Problem: Insufficient analysis support in existing compiler infrastructures due to non-transferability of analysis implementations

Decouples analysis algorithms from intermediate representations (IRs) by developing analysis-specific interfaces

Analysis reuse across compiler infrastructures

- Enable researchers to leverage prior work
- Enable direct comparisons amongst analyses
- Increase the impact of compiler analysis research

Software Architecture for OpenAnalysis



CS553 Lecture

Scanning and Parsing

19

Project 1: Basic Outline

- 1) Download and build OpenAnalysis
- 2) Copy Project1.tar to your CS directory and build
- 3) Implement 3 parsers that build up certain parts of a subsidiary IR using the examples in testSubIR.cpp and Input/testSubIR.oa
- 4) Next week start testing FIAlias implementation in OpenAnalysis

CS553 Lecture

Scanning and Parsing

20

Concepts

Compilation stages in a compiler

- Scanning, parsing, semantic analysis, intermediate code generation, optimization, code generation

Lexical analysis or scanning

- Tools: lex, flex, etc.

Syntactic analysis or parsing

- Tools: yacc, bison, etc.

Next Time

Lecture

- Undergrad compilers in a day!