Lesson 3

• Overview of a Simple Compiler Front-end
  – Lexical analysis
  – Intermediate code generation
  – Static checking
Admins

• Office Hours:
  – Wednesday, 9 - 11

• Check your data and add the University ID (matricola) in the sheet
Compiler Front- and Back-end

Source program (character stream)

Scanner (lexical analysis)

Tokens

Parser (syntax analysis)

Parse tree

Semantics Analysis

Abstract syntax tree, or …

Intermediate Code Generation

Three address code, or …

Machine-Independent Code Improvement

Modified intermediate form

Target Code Generation

Assembly or object code

Machine-Specific Code Improvement

Modified assembly or object code
A Translator for Simple Expressions based on Predictive Parsing and Semantic Actions

\[
\begin{align*}
expr & \rightarrow expr + term \quad \{ \text{print("+") } \} \\
expr & \rightarrow expr - term \quad \{ \text{print("-") } \} \\
expr & \rightarrow term \\
term & \rightarrow 0 \quad \{ \text{print("0") } \} \\
term & \rightarrow 1 \quad \{ \text{print("1") } \} \\
\ldots & \quad \ldots \\
term & \rightarrow 9 \quad \{ \text{print("9") } \}
\end{align*}
\]

After left recursion elimination:

\[
\begin{align*}
expr & \rightarrow term \text{ rest} \\
rest & \rightarrow + term \quad \{ \text{print("+") } \} \text{ rest} \\
rest & \rightarrow - term \quad \{ \text{print("+") } \} \text{ rest} \\
rest & \rightarrow \varepsilon \\
term & \rightarrow 0 \quad \{ \text{print("0") } \} \\
term & \rightarrow 1 \quad \{ \text{print("1") } \} \\
\ldots & \\
term & \rightarrow 9 \quad \{ \text{print("9") } \}
\end{align*}
\]
main()
{    lookahead = getchar();
    expr();
}
expr()
{   term();
    while (1) /* optimized by inlining rest() and removing recursive calls */
    {   if (lookahead == '+')
        {   match('+'); term(); putchar('+');
        }
    else if (lookahead == '-')
        {   match('-'); term(); putchar('-');
        }
    else break;
    }
}
term()
{   if (isdigit(lookahead))
    {   putchar(lookahead); match(lookahead);
    }
    else error();
}
match(int t)
{   if (lookahead == t)
        lookahead = getchar();
    else error();
}
error()
{   printf("Syntax error\n");
    exit(1);
}
The Structure of the Front-End

Source Program (Character stream) → Lexical analyzer → Token stream → Syntax-directed translator → Intermediate representation

Develop parser and code generator for translator

Syntax definition (BNF grammar) → IR specification
Adding a Lexical Analyzer

• Typical tasks of the lexical analyzer:
  – Remove white space and comments
  – Encode constants as tokens
  – Recognize keywords
  – Recognize identifiers and store identifier names in a global symbol table
The Lexical Analyzer ("lexer")

\[ y := 31 + 28 \times x \]

Parser

The lookahead of the Parser can be a token, not just a character
Token Attributes

The parser accesses the token via `lookahead`, and the token attribute via the global variable `tokenval`

\[ factor \rightarrow ( \text{expr} ) \]
\[ \quad \mid \text{num} \{ \text{print(num.value)} \} \]

#define NUM 256 /* token returned by lexer */

factor()
{
    if (lookahead == '(')
    {
        match('('); expr(); match(')');
    }
    else if (lookahead == NUM)
    {
        printf(" %d ", tokenval); match(NUM);
    }
    else error();
}

Symbol Table

The symbol table is globally accessible (to all phases of the compiler)

Each entry in the symbol table contains a string and a token value:

```c
struct entry
{
    char *lexptr; /* lexeme (string) for tokenval */
    int token;
};

struct entry symtable[];

insert(s, t): returns array index to new entry for string s token t
lookup(s): returns array index to entry for string s or 0
```

Possible implementations:
- simple C code
- hashtables
Handling identifiers (lexer)

Code executed by the lexer after an identifier has been recognized (stored in `lexbuf`):

```c
/* lexer.c */
int lexan()
{
    ...  
    tokenval = lookup(lexbuf);
    if (tokenval == 0) /* not found */
        tokenval = insert(lexbuf, ID);
    return symtable[tokenval].token;
}
```
Handling identifiers (parser)

\[
\text{factor } \rightarrow ( \text{ expr } ) \\
\quad | \text{id} \{ \text{print(id.string)} \}
\]

#define ID 259 /* token returned by lexer */

factor()
{
  if (lookahead == '(
  {
    match(('; expr(); match(')));
  }
  else if (lookahead == ID)
  {
    printf(" %s ", symtable[tokenval].lexptr);
    match(ID);
  }
  else error();
}
Handling Reserved Keywords (lexer)

```c
/* global.h */
#define DIV 257 /* token */
#define MOD 258 /* token */
#define ID 259 /* token */

/* init.c */
insert("div", DIV);
insert("mod", MOD);

/* lexer.c */
int lexan()
{
    ...
    tokenval = lookup(lexbuf);
    if (tokenval == 0) /* not found */
        tokenval = insert(lexbuf, ID);
    return symtable[tokenval].token;
}
```

Simply initialize the global symbol table with the set of keywords
Handling Reserved Keywords (parser)

morefactors → \texttt{div} \texttt{factor} \{ \texttt{print( ‘DIV’ )} \} morefactors
| \texttt{mod} \texttt{factor} \{ \texttt{print( ‘MOD’ )} \} morefactors
| …

/* parser.c */
morefactors()
{   if (lookahead == DIV)
    {   match(DIV); factor(); printf(“DIV”); morefactors();
    }
  else if (lookahead == MOD)
    {   match(MOD); factor(); printf(“MOD”); morefactors();
    }
  else
    ...
}
Symbol Tables and Scopes

• The same identifier can be declared several times in a program (e.g. in different blocks)
• Each declaration has its own attributes (e.g. type)
• A solution: one Symbol Table per scope
  – Chain of symbol tables for nested blocks
  – Hash table + auxiliary stack
  – Entries have to be created by the parser
The Structure of the Front-End

Source Program (Character stream) → Lexical analyzer → Token stream → Syntax-directed translator → Intermediate representation

Syntax definition (BNF grammar)

IR specification

Develop parser and code generator for translator
Intermediate Code Generation

• Two main kinds of intermediate representations:
  – Trees (parse trees, abstract syntax trees)
    • Useful for static semantic analysis (“static checking”)
  – Linear representations (“three-address code”)
    • Good for machine-independent optimization

• Often compilers produce the linear code during on-the-fly generation of the syntax tree
Translation Scheme for generating the Abstract Syntax Tree: Expressions

- Each operator is a node, with “semantically meaningful components” as children
- For each production the semantic action either builds a new node (with suitable parameters), or returns the node of the only subexpression

<table>
<thead>
<tr>
<th>Production</th>
<th>Rule 1</th>
<th>Rule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr → rel = expr₁</td>
<td>rel = new Assign('=', rel.n, expr₁.n); }</td>
<td>expr.n = new Assign('=', rel.n, expr₁.n); }</td>
</tr>
<tr>
<td></td>
<td>rel</td>
<td>expr.n = rel.n; }</td>
</tr>
<tr>
<td>rel → rel₁ &lt; add</td>
<td>rel₁ &lt; add</td>
<td>rel₁ &lt; new Rel('&lt;', rel₁.n, add.n); }</td>
</tr>
<tr>
<td></td>
<td>rel₁ &lt;= add</td>
<td>rel₁ &lt;= new Rel('&lt;=', rel₁.n, add.n); }</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>add.n = new Rel('&lt;', rel₁.n, add.n); }</td>
</tr>
<tr>
<td>add → add₁ + term</td>
<td>add₁ + term</td>
<td>add₁ + new Op('+', add₁.n, term.n); }</td>
</tr>
<tr>
<td></td>
<td>term</td>
<td>add.n = new Op('+', add₁.n, term.n); }</td>
</tr>
<tr>
<td></td>
<td>term</td>
<td>add.n = term.n; }</td>
</tr>
<tr>
<td>term → term₁ * factor</td>
<td>term₁ * factor</td>
<td>term₁ * new Op('*', term₁.n, factor.n); }</td>
</tr>
<tr>
<td></td>
<td>factor</td>
<td>term.n = new Op('*', term₁.n, factor.n); }</td>
</tr>
<tr>
<td>factor → ( expr )</td>
<td>( expr )</td>
<td>factor.n = new Num(num.value); }</td>
</tr>
<tr>
<td></td>
<td>num</td>
<td>factor.n = expr.n; }</td>
</tr>
</tbody>
</table>
Translation Scheme for generating the Abstract Syntax Tree: Statements

• Statements as operators: note that the concrete syntax is dropped

\[
\begin{align*}
program & \rightarrow \ block & & \{ \text{return block.n; } \} \\
block & \rightarrow \ '{\} \ stmts '{\}' & & \{ \text{block.n = stmts.n; } \} \\
stmts & \rightarrow \ stmts_1 \ stmt & & \{ \text{stmts.n = new Seq(stmts_1.n, stmt.n); } \} \ \\
& & | \epsilon & \{ \text{stmts.n = null; } \} \\
stmt & \rightarrow \ expr ; & & \{ \text{stmt.n = new Eval(expr.n); } \} \ \\
& & | \ \text{if ( expr ) stmt}_1 & \{ \text{stmt.n = new If(expr.n, stmt_1.n); } \} \ \\
& & | \ \text{while ( expr ) stmt}_1 & \{ \text{stmt.n = new While(expr.n, stmt_1.n); } \} \ \\
& & | \ \text{do stmt}_1 \ \text{while ( expr )} & \{ \text{stmt.n = new Do(stmt_1.n, expr.n); } \} \ \\
& & | \ block & \{ \text{stmt.n = block.n; } \}
\end{align*}
\]
An example: from a statement to the abstract syntax tree

```
<if> <(> <id, "peek"> <eq> <const, '\n'> <>)>
  <id, "line"> <assign> <id, "line"> <+> <num, 1> <;
```
if ( peek == '\n' ) line = line + 1;
Static Checking

- **Syntactic properties** (not captured by the context-free grammar of the language) are checked by analyzing the parse tree or the abstract syntax tree.

- **Context-dependent syntactic** properties:
  1. Every variable is declared before used.
  2. Each identifier is declared at most once per scope.
  3. Left operands of assignments are L-values.
  4. Break statements must have enclosing loop or switch.

- **Semantic analysis** is applied by the compiler to discover the meaning of a program by analyzing its parse tree or abstract syntax tree. Useful to prevent runtime errors.

- **Static semantic** checks performed at compile time:
  5. Type checking: each operator is applied to arguments of the right type.
     - Handling of coercion and overloading.
Exercise

• For each of the numbered items in the last slide, discuss how the property can be checked either with a translation scheme or with suitable attributes of the parse tree
Semantic Analysis

• Dynamic semantic checks are performed at run time, and the compiler produces code that performs these checks
  – Array subscript values are within bounds
  – Arithmetic errors, e.g. division by zero
  – Pointers are not dereferenced unless pointing to valid object
  – A variable is used but hasn't been initialized
  – When a check fails at run time, an exception is raised
Generation of Three Address Code

• Linear Intermediate Representation generated by structural induction executing a function on the nodes of the tree

• Sequence of instructions of the form
  \[ x = y \text{ op } z \]

• Arrays handled with instructions
  \[ x[y] = z \]
  \[ x = y[z] \]

• Sequence control handled with jump instructions:
  \[ \text{ifFalse } x \text{ goto } L \]
  \[ \text{ifTrue } x \text{ goto } L \]
  \[ \text{goto } L \]

• Statement may have any number of labels, e.g.
  \[ L1:L2 : x = y \]
Translation of Statements

• Jumps are used to implement the control flow
• Example: if expr then stmt₁ is translated to

class If extends Stmt {
  Expr E; Stmt S;
  public If(Expr x, Stmt y) { E = x; S = y; after = newlabel(); }
  public void gen() {
    Expr n = E.rvalue();
    emit("ifFalse" + n.toString() + " goto " + after);
    S.gen();
    emit(after + ":");
  }
}

| code to compute expr into x |
| ifFalse x goto after |
| code for stmt₁ |
| after |
Translation of expressions

• One operation at a time, using temporaries

\[ i - j + k \rightarrow \begin{cases} t1 = i - j \\ t2 = t1 + k \end{cases} \]

\[ 2 * a[i] \rightarrow \begin{cases} t1 = a[i] \\ t2 = 2 * t1 \end{cases} \]

• L-values in assignments cannot be translated into temporaries

\[ a[i] = 2 * a[j - k] \rightarrow \begin{cases} t3 = j - k \\ t2 = a[t3] \\ t1 = 2 * t2 \\ a[i] = t1 \end{cases} \]
Translation of expressions: the pseudocode for L-value and R-value

Expr lvalue(x : Expr) {
    if ( x is an Id node ) return x;
    else if ( x is an Access(y, z) node and y is an Id node ) {
        return new Access(y, rvalue(z));
    }
    else error;
}

Expr rvalue(x : Expr) {
    if ( x is an Id or a Constant node ) return x;
    else if ( x is an Op(op, y, z) or a Rel(op, y, z) node ) {
        t = new temporary;
        emit string for t = rvalue(y) op rvalue(z);
        return a new node for t;
    }
    else if ( x is an Access(y, z) node ) {
        t = new temporary;
        call lvalue(x), which returns Access(y, z');
        emit string for t = Access(y, z');
        return a new node for t;
    }
    else if ( x is an Assign(y, z) node ) {
        z' = rvalue(z);
        emit string for lvalue(y) = z';
        return z';
    }
    }
}
## The Phases of a Compiler

<table>
<thead>
<tr>
<th>Phase</th>
<th>Output</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programmer (source code producer)</strong></td>
<td>Source string</td>
<td>\texttt{A=B+C;}</td>
</tr>
<tr>
<td><strong>Scanner (performs lexical analysis)</strong></td>
<td>Token string</td>
<td>\texttt{‘A’, ‘=’, ‘B’, ‘+’, ‘C’, ‘;’} And symbol table with names</td>
</tr>
<tr>
<td><strong>Parser (performs syntax analysis</strong></td>
<td>Parse tree or abstract syntax tree</td>
<td></td>
</tr>
<tr>
<td>based on the grammar of the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>programming language**</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Semantic analyzer (type checking, etc)</strong></td>
<td>Annotated parse tree or abstract syntax tree</td>
<td></td>
</tr>
<tr>
<td><strong>Intermediate code generator</strong></td>
<td>Three-address code, quads, or RTL (Register Transfer Language)</td>
<td>\texttt{int2fp B} \texttt{t1} + \texttt{t1} \texttt{C} \texttt{t2} := \texttt{t2} \texttt{A}</td>
</tr>
<tr>
<td><strong>Optimizer</strong></td>
<td>Three-address code, quads, or RTL</td>
<td>\texttt{int2fp B} \texttt{t1} + \texttt{t1} \texttt{#2.3} \texttt{A}</td>
</tr>
<tr>
<td><strong>Code generator</strong></td>
<td>Assembly code</td>
<td>\texttt{MOVF #2.3,r1} \texttt{ADDF2 r1,r2} \texttt{MOVF r2,A}</td>
</tr>
<tr>
<td><strong>Peephole optimizer</strong></td>
<td>Assembly code</td>
<td>\texttt{ADDF2 #2.3,r2} \texttt{MOVF r2,A}</td>
</tr>
</tbody>
</table>
Reviewing the Entire Process

\[
\text{position} := \text{initial} + \text{rate} \times 60
\]

lexical analyzer

\[
\text{id1} := \text{id2} + \text{id3} \times 60
\]

syntax analyzer

\[
:= \quad + \quad \cdot \quad 60
\]

semantic analyzer

\[
\text{id1} := \text{id2} + \text{id3} \times \text{inttoreal} \times 60
\]

intermediate code generator

Symbol Table

position ....
initial ....
rate....

Errors
Reviewing the Entire Process

Symbol Table
position ....
initial ....
rerate....

intermediate code generator

* t1
temp3 := id2 + t2
id1 := t3

MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R1, R2
MOVF R1, id1

Errors

code optimizer
t1 := id3 * 60.0
id1 := id2 + t1

final code generator

3 address code

t1 := inttoreal(60)
t2 := id3 * t1
temp3 := id2 + t2
id1 := t3