Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 4

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

Front end analysis

Source program (character stream)

- Scanner (lexical analysis)
  - Tokens
    - Parser (syntax analysis)
      - Parse tree
        - Semantic Analysis
          - Abstract syntax tree, or…
            - Intermediate Code Generation
              - Three address code, or…

Back end synthesis

- 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

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

After left recursion elimination:

\[
\begin{align*}
expr & \rightarrow term\ rest \\
rest & \rightarrow +\ term \quad \{ \text{print(“+”) } \} \ rest \\
rest & \rightarrow -\ term \quad \{ \text{print(“+”) } \} \ rest \\
rest & \rightarrow \varepsilon \\
term & \rightarrow 0 \quad \{ \text{print(“0”) } \} \\
term & \rightarrow 1 \quad \{ \text{print(“1”) } \} \\
\quad \ldots \quad \ldots \\
term & \rightarrow 9 \quad \{ \text{print(“9”) } \}
\end{align*}
\]
Code of the translator

main()
{
  lookahead = getchar();
  expr();
}

expr()
{
  term(); rest();
}

rest()
{
  if (lookahead == '+')
  {
    {match('+'); term(); putchar('+'); rest();
    }
    else if (lookahead == '-')
    {
      {match('-'); term(); putchar('-'); rest();
    }
    else {};
  }

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);
}
Optimized code of the translator

\[
\begin{align*}
expr & \rightarrow term \ rest \\
rest & \rightarrow + \ term \ { \{ \text{print(“+”) } \} \ rest} \\
rest & \rightarrow - \ term \ { \{ \text{print(“-”) } \} \ rest} \\
rest & \rightarrow \varepsilon \\
term & \rightarrow 0 \ { \{ \text{print(“0”) } \} } \\
term & \rightarrow 1 \ { \{ \text{print(“1”) } \} } \\
& \ldots \\
term & \rightarrow 9 \ { \{ \text{print(“9”) } \} }
\end{align*}
\]
The Structure of the Front-End

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

- Lexical analyzer
- Syntax-directed translator
- 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")

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

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

The parser accesses the token via \texttt{lookahead}, and the token attribute via the global variable \texttt{tokenval}.

\[
\text{id} \rightarrow ( \text{expr} ) \\
\quad | \quad \text{num} \{ \text{print(num.value)} \}
\]

\#define \texttt{NUM} 256 /* token returned by lexer */

\text{factor}() /* code of the parser */
{  
  if (\texttt{lookahead} == '(')  
  { \text{match('('); expr(); match(')');} 
  }  
  else if (\texttt{lookahead} == \texttt{NUM})  
  { \text{printf(" %d ", tokenval); match(NUM);}  
  }  
  else \text{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
- hash tables
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)

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

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

factor()
{
    if (lookahead == '(
        
    }
    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 → div factor { print( ‘DIV’ ) } morefactors 
  | mod factor { 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

Develop parser and code generator for translator

Syntax definition (BNF grammar) → IR specification
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
Abstract Syntax Trees vs. Parse Trees

- **Parse Tree**: tree representation of *concrete syntax* of a program
  - Brackets are dropped
  - Keywords are dropped, constructs represented by nodes

- **AST**: tree representation of *abstract syntax* of program

\[
\begin{align*}
expr & \rightarrow expr + term \mid term \\
\text{term} & \rightarrow \text{term} \ast \text{factor} \mid \text{factor} \\
\text{factor} & \rightarrow \text{number} \mid ( expr )
\end{align*}
\]
Translation Scheme for generating the AST during parsing: Expressions

- Each node of the Parse Tree “points” to a node of the AST
- Each operator is a node of the AST, with “semantically meaningful components” as children
- Semantic actions either build a new node of the AST (with suitable parameters), or return the node of the only subexpression

\[
\begin{align*}
\text{expr} & \rightarrow \text{rel} = \text{expr}_1 \\
& \quad | \quad \text{rel} \\
& \quad | \quad \text{expr}_1 < \text{add} \\
& \quad | \quad \text{rel}_1 \leq \text{add} \\
& \quad | \quad \text{add} \\
\text{rel} & \rightarrow \text{rel}_1 < \text{add} \\
& \quad | \quad \text{rel}_1 \leq \text{add} \\
& \quad | \quad \text{add} \\
\text{add} & \rightarrow \text{add}_1 + \text{term} \\
& \quad | \quad \text{term} \\
\text{term} & \rightarrow \text{term}_1 \ast \text{factor} \\
& \quad | \quad \text{factor} \\
\text{factor} & \rightarrow ( \text{expr} ) \\
& \quad | \quad \text{num}
\end{align*}
\]

\{ \text{expr} . n = \text{new Assign('=', \text{rel} . n, \text{expr}_1 . n); } \}
\{ \text{expr} . n = \text{rel} . n; \}
\{ \text{rel} . n = \text{new Rel('<', \text{rel}_1 . n, \text{add} . n); } \}
\{ \text{rel} . n = \text{new Rel('\leq', \text{rel}_1 . n, \text{add} . n); } \}
\{ \text{rel} . n = \text{add} . n; \}
\{ \text{add} . n = \text{new Op('+', \text{add}_1 . n, \text{term} . n); } \}
\{ \text{add} . n = \text{term} . n; \}
\{ \text{term} . n = \text{new Op('*', \text{term}_1 . n, \text{factor} . n); } \}
\{ \text{term} . n = \text{factor} . n; \}
\{ \text{factor} . n = \text{expr} . n; \}
\{ \text{factor} . n = \text{new Num(\text{num} . value); } \}

Useless nonterminals are not represented

Drop brackets
Translation Scheme for generating the Abstract Syntax Tree: Statements

- Statements as operators
- The concrete syntax (keywords) is dropped

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

Parser

Scanner

Generation of Abstract Syntax Tree

```java
if ( peek == '\n' ) line = line + 1;
```

```xml
<if> <(> <id, "peek"> <eq> <const, '\n'> <)> <id, "line"> <assign> <id, "line"> <+> <num, 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:
  - 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
  \[ x = y \ op \ z \]
• Arrays handled with instructions
  \[ x[y] = z \]
  \[ x = y[z] \]
• Sequence control handled with jump instructions:
  \[ \text{ifFalse } x \ \text{goto } L \]
  \[ \text{ifTrue } x \ \text{goto } L \]
  \[ \text{goto } L \]
• Statements 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 + ":");
    }
}
Translation of expressions

• One operation at a time, using temporaries

\[ i - j + k \rightarrow \begin{cases} t_1 = i - j \\ t_2 = t_1 + k \end{cases} \]

\[ 2 \times a[i] \rightarrow \begin{cases} t_1 = a[i] \\ t_2 = 2 \times t_1 \end{cases} \]

• L-values in assignments cannot be translated into temporaries

\[ a[i] = 2 \times a[j - k] \rightarrow \begin{cases} t_3 = j - k \\ t_2 = a[t_3] \\ t_1 = 2 \times t_2 \\ a[i] = t_1 \end{cases} \]
Reviewing the Entire Process

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

- **lexical analyzer**
  \[
  \text{id1} := \text{id2} + \text{id3} \times 60
  \]

- **syntax analyzer**

- **semantic analyzer**

- **intermediate code generator**

- **Symbol Table**
  - position .......
  - initial ......
  - rate....

- **Errors**
Reviewing the Entire Process

Symbol Table
position .... initial .... rate....

intermediate code generator

\[
\begin{align*}
t1 & := \text{inttoreal}(60) \\
t2 & := \text{id3} \times t1 \\
temp3 & := \text{id2} + t2 \\
id1 & := t3
\end{align*}
\]

code optimizer

\[
\begin{align*}
t1 & := \text{id3} \times 60.0 \\
id1 & := \text{id2} + t1
\end{align*}
\]

final code generator

\[
\begin{align*}
\text{MOVF id3, R2} \\
\text{MULF #60.0, R2} \\
\text{MOVF id2, R1} \\
\text{ADDF R1, R2} \\
\text{MOVF R1, id1}
\end{align*}
\]