Structure of the Compiler

Mooly Sagiv
Tel Aviv University
sagiv@math.tau.ac.il
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

31. Oktober 2007

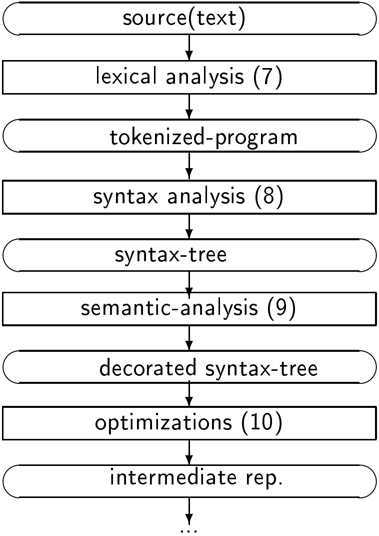
Subjects

- Structure of the compiler
- Automatic Compiler Generation
- Real Compiler Structures

Motivation

- The compilation process is decomposable into a sequence of tasks.
  Aspects:
  - Modularity
  - Reusabilty
- The functionality of the tasks is well defined.
- The programs that implement some of the tasks can be **automatically** generated from formal specifications

"Standard" Structure and implementing devices



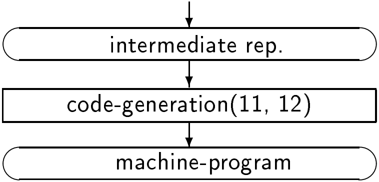
finite automata

pushdown automata

attribute grammar evaluators

abstract interpretation + transformations

"Standard" Structure cont'd



tree automata + dynamic programming + $\cdots$

A Running Example

program foo ;
var i, j : **real** ;
begin
    read (i);
    $j := i + 3 * i$
end.

Lexical Analysis (Scanning)

- Functionality

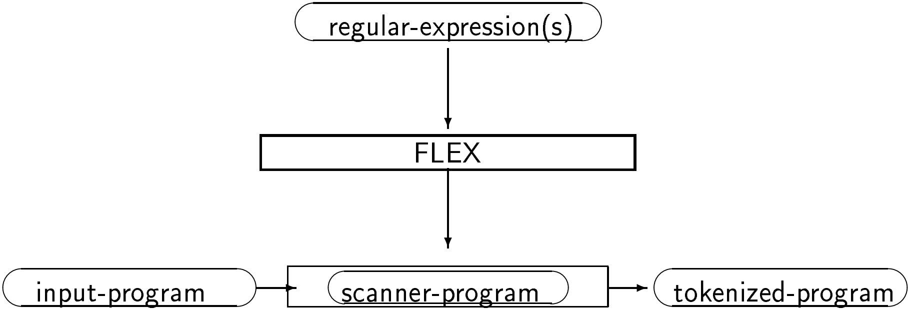
  Input program text as sequence of characters

  Output program text as sequence of symbols (tokens)
- Read input file
- Report errors about symbols illegal in the programming language
- **Screening** subtask:
  - Identify language keywords and standard identifiers
  - Eliminate "white-space", e.g., consecutive blanks and newlines
  - Count line numbers

Automatic Generation of Lexical Analyzers

- The symbols of programming languages can be specified by regular expressions.
- Examples:
  - `program` as a sequence of characters.
  - `(alpha (alpha | digit)*)` for Pascal identifiers
  - “`(*`“ `until` “`*)`“ for Pascal comments
- The recognition of input strings can be performed by a **finite automaton**.
- A table representation or a program for the automaton is **automatically generated** from a **regular** expression.

Automatic Generation of Lexical Analyzers (cont'd)

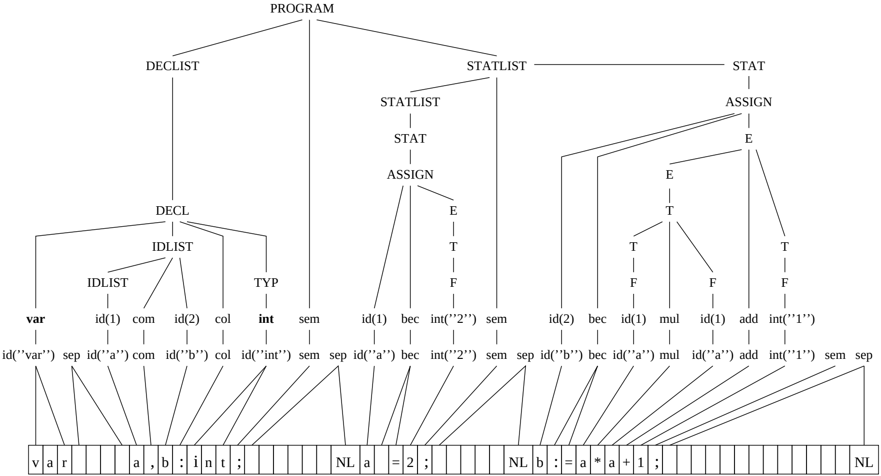


Numerous generators for lexical analyzers: lex, flex, oolex, quex, ml-lex.

Syntax Analysis (Parsing)

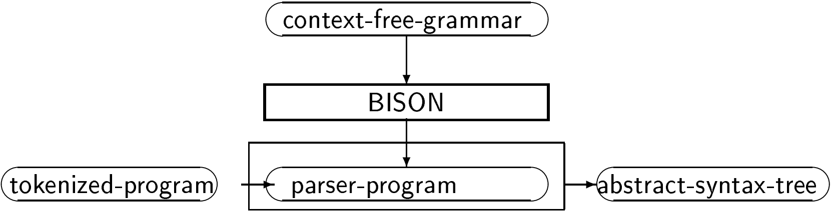
- Functionality
  - **Input** Sequence of symbols (tokens)
  - **Output** Structure of the program:
    - concrete syntax tree (parse tree),
    - abstract syntax tree, or
    - derivation.
- Treat syntax errors
  - **Report** (as many as possible) syntax errors,
  - **Diagnose** syntax errors,
  - **Correct** syntax errors.

Parse Tree


PROGRAM
DECLIST
STATLIST
STAT
STATLIST
ASSIGN
STAT
E
ASSIGN
E
T
DECL
E
T
IDLIST
T
T
IDLIST
TYP
F
F
F
F
var
id(1)
com
id(2)
col
int
sem
id(1)
bec
int(''2'')
sem
id(2)
bec
id(1)
mul
id(1)
add
int(''1'')
id(''var'')
sep
id(''a'')
com
id(''b'')
col
id(''int'')
sem
sep
id(''a'')
bec
int(''2'')
sem
sep
id(''b'')
bec
id(''a'')
mul
id(''a'')
add
int(''1'')
sem
sep
v a r a , b : i n t ; NL a = 2 ; NL b : = a * a + 1 ; NL

Automatic Generation of Syntax Analysis

- Parsing of programs can be performed by a **pushdown automaton**.
- A table representation or a program for the pushdown automaton is **automatically generated** from a context free grammar.



Numerous parser generators: yacc, bison, ml-yacc, java-CC, ANTLR.

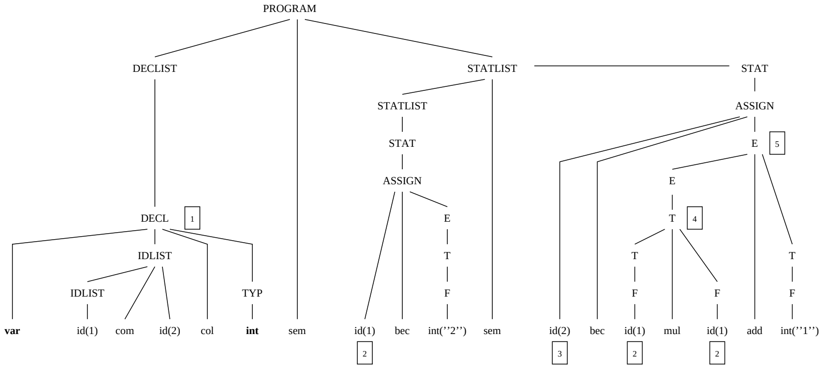
Semantic Analysis

- Functionality

  Input Abstract syntax tree

  Output Abstract tree “decorated“ with attributes, e.g., types of sub-expressions
- Report “semantic“ errors, e.g., undeclared variables, type mismatches
- Resolve usages of variables:
  Identify the right defining occurrences of variables for applied occurrences.
- Compute type of every (sub-)expression, resolving overloading.

Decorated parse tree



1	(id(1),(var,int)) (id(2),(var,int))	2	(var,int)	3	(var,int)	4	int	5	int
1	(id(1),(var,int,0)) (id(2),(var,int,0))	2	(var,int,0)	3	(var,int,1)				

Machine Independent Optimizations

- Functionality
  - Input: Abstract tree decorated with attributes
  - Output: A semantically equivalent abstract tree decorated with attributes
- Analyzes the program for global properties.
- Transforms the program based on these global properties in order to improve efficiency.
- Analysis may also report program anomalies, e.g., uninitialized variables.

Example1: Constant Propagation

```
const i = 5;
var x, y : integer;
begin
    x := 5 + i ;
    read y;
    if x = y
    then y := y + x
    else y := y − x
    fi;
    y := y + x ∗ 9
end;
```

Example2: Loop Invariant Code Motion and Reduction in Operator Strength

const $i = 5$;
var n, x, y : **integer**;
begin
    $x := 5 + i$;
    $y := 1$;
    read n;
    for $k := 1$ **to** 100 **do**
        $y := y + k \times (x + n)$
    od;
    print y
end;

Address Assignment

- Map variables into the static area, stack, heap
- Compute static sizes
- Generate proper **alignments**

Generation of the target program

Partly contradictory goals:

- **Code Selection**: Select cheapest instruction sequence.
- **Register Allocation**: Perform most or all of the computations in registers.
- **Instruction Scheduling**: On machines with intraprocessor parallelism, e.g., super-scalar, pipelined, VLIW: exploit intraprocessor parallelism as much as possible.
- Partial problems are already NP-hard.
- "Good" solutions are obtained by combining suboptimal solutions obtained by heuristics

Example: Local Register Allocation

- Try to perform all computations in registers:
- One register is sufficient for the (trivial) expression x; so execute the command:

$$load\ r_i, \rho(x)$$

- If the expression e_1 takes m registers to evaluate and e_2 takes n registers and $m > n$, then $e_1 + e_2$ takes m registers (why?)
- If the expression e_1 takes m registers and e_2 takes n registers and $m < n$, then $e_1 + e_2$ takes n registers (why?)
- What happens if $m = n$?
- What happens if there aren't enough registers?

Real Compiler Structure

- Simple compilers are “one-pass”; conceptually separated tasks are combined.
  Parser is the **driver**.
- One task in the conceptual compiler structure may need more than one pass, e.g., mixed declarations and uses.
- Almost all use automatically generated lexers and parsers.
- Compilers use global information, e.g., symbol tables.
- There may be many representation levels in a multipass compiler.