Compilers

Compiler Construction Tutorial
The Front-end

Salahaddin University
College of Engineering
Software Engineering Department
2011-2012

Amanj Sherwany
http://www.amanj.me/wiki/doku.php?id=teaching:su:compilers

Introduction

- In this tutorial we go through the implementation of *dragonfront* compiler which is discussed in the Purple Dragon Book.
- The source code can be found in the webpage of the course.
- There are many more ways to implement compilers, this is just an example.

The CFG of the Language

- *program* → *block*
- *block* → { *decls stmts* }
- *decls* → *decls decl* | ε
- *decl* → *type* **id** ;
- *type* → *type* [**num**] | **basic**
- *stmts* → *stmts stmt* | ε
- *stmt* → *loc* = *bool* ;
  | **if** (*bool*) *stmt*
  | **if** (*bool*) *stmt* **else** *stmt*
  | **while** (*bool*) *stmt*
  | **do** *stmt* **while** (*bool*) ;
  | **break** ;
  | *block*
- *loc* → *loc* [*bool*] | **id**

The CFG of the Language, *Cont'd*

- *bool* → *bool* | | *join* | *join*
- *join* → *join* && *equality* | *equality*
- *equality* → *equality* == *rel* | *equality* != *rel* | *rel*
- *rel* → *expr* < *expr* | *expr* <= *expr* | *expr* >= *expr* | *expr* > *expr* | *expr*
- *expr* → *expr* + *term* | *expr* – *term* | *term*
- *term* → *term* * *unary* | *term* / *unary* | *unary*
- *unary* → ! *unary* | - *unary* | *factor*
- *factor* → (*bool*) | *loc* | **num** | **real** | **true** | **false**

Packages

- The Java code for the translator consists of five packages:
  - *main*: here the execution starts.
  - *lexer*: the lexical analyzer is here.
  - *symbols*: implements symbol tables and types.
  - *parser*: parsing is done here
  - *inter*: contains classes for the language constructs in the abstract syntax. (intermediate code)

The Classes in “lexer”

- *Tag*: defines constants for tokens
  - INDEX, MINUS and TEMP are not lexical tokens; they will be used in syntax trees.
- *Token*: is a parent class for the possible tokens
- *Num*: represents integer numbers
- *Word*: represents *reserved words*, *identifiers* and *composite tokens* like *&&*.
  - It is also useful for managing the written form of operations in the intermediate code like unary minus; for example the source text **-2** has the intermediate form **minus 2**.

The Classes in "lexer", *Cont'd*

- *Real*: is for floating point numbers
- *Lexer*: the core class of the lexical analyzer
  - The most important method here is *scan*, which recognizes:
    - Numbers
    - Identifiers
    - Reserved words
  - Function **readch()** is used to read the next input character into variable **peek**
  - Function **scan()** scans the input and recognizes the possible tokens

The Classes in “symbols”

- *Env*: represents environments (symbol table), and maps word tokens to objects of class *Id*, which is defined in package *inter* a long with the classes for expressions and statements.
- *Type*: is a subclass of *Word*, since basic type names like **int** are basically reserved words:
  - The objects for the basic types are **Type.Int**, **Type.Float**, **Type.Char** and **Type.Bool**
  - All of them have inherited field **tag** set to **Tag.BASIC**, so the parser treats them all alike.

The Classes in "symbols", *Cont'd*

- In *Type* class, functions **numeric** and **max** are useful for type conversions.
  - Conversions are allowed between the "numeric" types **Type.Char**, **Type.Int**, and **Type.Float**
  - When an arithmetic operator is applied to two numeric types, the result has the "max" of the two types.
- Arrays are the only constructed type in the source language

The Classes in “inter”

- The package *inter* contains the *Node* class hierarchy, *Node* has two subclasses: *Expr* for expression nodes and *Stmt* for statement nodes.
- Nodes in the syntax tree are implemented as objects of class *Node*.
- For error reporting, filed **lexline** saves the source-line number of the construct at this node.
- Expression constructs are implemented by subclasses of *Expr*.

The Classes in "inter", *Cont'd*

- Class *Expr* has fields **op** and **type**, representing the operator and type, respectively at a node.
- Method **gen** returns a "term" that can fit the right side of a three address instruction:
  - Given expression $E = E_1 + E_2$, method **gen** returns a term x_1+x_2, where x_1 and x_2 are addresses for the values of E_1 and E_2, respectively.
  - The return value **this** is appropriate if this object is an address; subclasses of *Expr* typically reimplement **gen**.

The Classes in "inter", *Cont'd*

- Method **reduce** computes or "reduces" an expression down to a single address;
  - That is, it returns a constant, an identifier, or a temporary name.
    - Given expression *E*, method **reduce** returns a temporary *t* holding the value of *E*.
    - Again, **this** is an appropriate return value if this object is an address.

Subclasses of Expr

- *Id* inherits the default implementation of **gen** and **reduce** in class *Expr*, since an identifier is an address.
  - The node for an identifier of class *Id* is a leaf.
  - The call **super(id, p)** saves id and p in inherited fields **op** and **type**, respectively.
  - Field **offset** holds the relative address of this identifier.

Subclasses of Expr

- Class *Op* provides an implementation of **reduce** that is inherited by subclasses *Arith* for arithmetic operators, *Unary* for unary operators and *Access* for array accesses.
  - In each case, **reduce** calls **gen** to generate a term, emits an instruction to assign the term to a new temporary name, and returns the temporary.

Subclasses of Expr, *Cont'd*

- *Arith* implements binary operators like + and *.
  - Constructor *Arith* begins by calling **super(tok,null)**, where **tok** is a token representing the operator and **null** is a placeholder for the type.
  - The type is determined by using **Type.max**, which checks whether the two operations can be coerced to a common numeric type.
  - This simple compiler checks types, but does not insert type conversions.

Subclasses of Expr, *Cont'd*

- In class *Arith*, method **gen** constructs the right side of a three-address instruction by reducing the subexpressions to addresses and applying the operator to the addresses.
- Class *Unary* is the one-operand counterpart of class *Arith*.
- Class *Temp*, represents the temporary variables (virtual registers).

Jumping Code for Boolean Expressions

- Jumping code for a boolean expression *B* is generated by method **jumping** (in *Expr*)
  - Takes two labels **t** and **f** as parameters, called the true and false exits of *B*, respectively.
  - The code contains jump to **t** if *B* evaluates to true, and a jump to **f** if *B* evaluates to false.
  - By convention, the special label 0 means that control falls through *B* to the next instruction after the code for *B*.
- *Constant* represents **True** and **False**
- *Rel* represents operators like: <, <=, ==, !=, >= and >

Intermediate Code for Statements

- Each statement construct is implemented by a subclass of *Stmt*.
- The fields for the components of a construct are in the relevant subclass;
  - For example class *While* has fields for a test expression and a substatement.
- The *Stmt* class (and its subclasses) deal with syntax-tree construction.
- **Stmt.Null** represents an empty sequence of statements.

Intermediate Code for Statements, *Cont'd*

- The method **gen** is called with two labels **b** and **a**, where **b** marks the beginning of the code for this statement and **a** marks the first instruction after the code for this statement.
- The subclasses *While* and *Do* save their label **a** in the field **after** so it can be used by any enclosed break statement to jump out of its enclosing construct.

Intermediate Code for Statements, *Cont'd*

- The object **Stmt.Enclosing** is used during parsing to keep track of the enclosing construct.
  - For a source language with **continue** statements, we can use the same approach to keep track of the enclosing construct for a **continue** statement.

Intermediate Code for Statements, *Cont'd*

- Class *Set* implements assignments with an identifier on the left side and an expression on the right.
- Class *SetElem* implements assignments to an array element.
- Class *Seq* implements a sequence of statements.

Parser

- The parser reads a stream of tokens and builds a syntax tree by calling the appropriate constructor functions.
- Class *Parser* has a procedure for each nonterminal.
    - The procedures are based on a grammar formed by removing left recursion from the source-language grammar.
- Parsing begins with a call to procedure **program**, which calls **block()** to parse the input stream and build the syntax tree.

Parser, *Cont'd*

- Symbol-table handling is shown explicitly in procedure **block**.
- Variable **top** holds the top symbol table
- Variable **savedEnv** is a link to the previous symbol table.
- Procedure **stmt** has a switch statement with cases corresponding to the productions for nonterminal *Stmt*.

Your Task

- There is *break* in the language, but there is no *continue*, your work is to add it.
- In the loops, there is no *for loop*, you have got to support it.
- Extend the given language to support *Switch*.
- Your compiler should read form text files.

Questions?