C++ Gold Parser Engine Documentation version 0.1

Manuel Astudillo

27th April 2003

1 Overview

The C++ Gold Parser Engine (GP Engine) is a library which implements a parser engine based on the grammars generated by Gold Parser builder.

The engine can be used to perform the first phase of a compilator i.e. performing the lexical and syntactical analysis and returning a *Reduction Tree* (RT). A RT is nothing more than a tree where every node represents a reduction of the source.

It is possible to use Gold Parser Builder, which has a test facility that compiles a source and returns its RT, to get a better understanding of what a reduction tree is.

Under compilator developing it is often desirable to transform the RT to an Abstract Syntax Tree. This library also provides a class that can be used to make this transformation a bit more easy.

In this piece of documentation we are going to explain briefly how to use the library. Although easy to use, a bit of knowledge in compilators technology is desirable.

2 API

The API is available at the home page of the library (cpp-gpengine.sourceforge.net) in form of doxygen automatic generated files.

3 Creating compilers

In this section an explanation of the different phases in the creation of a compilator are given.

3.1 Writting the grammar

The grammar has to be written using Gold Parser Builder. The latest version can be found at: `http://www.devincook.com`. In the same webpage is possible to find documentation on how to use the builder and how to write and test a grammar.

3.2 Designing the Abstract Syntax Tree

Designing an AST is pretty important before you can begin to code a class that transforms the reduction tree into the desired AST.

But why is so important to have an AST? In short the AST is just a tree which holds in a structured manner the relevant information from the source code. Having an AST is very convenient because from this structure it will be very easy to perform the next stages of the compiler, such as name analysis, type analysis, optimizations, code generation, etc.

Here we are going to give an example of how to write an abstract grammar from a simple grammar for a propositional logic:

```
<Start> ::= <RuleList>
```

```
<NewLines> ::= NEWLINE <NewLines>
             |

<RuleList> ::= <Rule> <NewLines> <RuleList>
             |

<Rule> ::= <Or> NEWLINE

<Or> ::= <And> OR <Or>
       | <And>

<And> ::= <Not> AND <And>
          | <Not>

<Not> ::= NOT <Implication>
          | <Implication>

<Implication> ::= <DoubleImplication> '->' <Implication>
                | <DoubleImplication>


<DoubleImplication> ::= <Value> '<->' <DoubleImplication>
                      | <Value>

<Value> ::= '(' <Rule> ')'
          | <Proposition>

<Proposition> ::= ID

! Definition of Terminals

{WS} = {Whitespace} - {CR} - {LF}
Whitespace = {WS}+

NEWLINE = {CR}{LF} | {LF}
ID = {Letter}{AlphaNumeric}*
```

The syntax to specify the abstract grammar uses extended BNF for easier legibility and compactness:

```
Start := [DefinitionSection] [RuleSection]

DefinitionSection := Definition*
Definition := Rule Rule

RuleSection := Rule*

Or:Rule := Rule Rule
And:Rule := Rule Rule
Not:Rule := Rule
```

```
Implication:Rule := Rule Rule
DoubleImplication:Rule := Rule Rule

Proposition:Rule := ID
```

As you can see the abstract grammar defines a nice tree with only the relevant information. Also it is very important to note the importance of the subclassing. Since the different Rules can be defined using different operations every operation has to be subclassed from the parent class Rule. The structure of this tree is also very suitable for more interesting things. As for example semantic analysis, interpretation, code generation, etc.

Another way to see the RT and AST paradigm is to think that the AST is what you actually want to have. While the grammar that generates the RT is the only way to express it in Gold Parser Builder. It can actually be recomendable to start defining the Abstract Grammar and afterwards try to represent it within the builder syntax.

3.3 From a RT to an AST

When you have succeded creating an AST that conveniently represents your grammar, a bit of code has to be written. Eventually it is possible to write this transform class/function from scratch but a helper class called ASTCreator is provided.

If you want to use this class then you have to make first a subclass of it and then follow the framework as it is explained in ASTCreator class. That is, generating a if sencente for every possible reduction and performing the correspondent transformation to the AST.

This procedure is actually not very difficult to do, but it can be a bit repetitive and boring. It is planed to implement a mechanism that can perform this transformation automatically, but at the moment it is just in early planning stages.

To make the transformation easier you should write the grammar carefully and make it simple and compact. If needed add more non-terminals with the only function of making the transformation easier.

3.4 An example

A script engine has been implemented using gold parser engine. This engine provides source code and documentation and can be used as a teaching example of how to use cpp-gpengine. The script engine can be located at: `www.efd.lth.se/~d00mas/lance/`.

4 Further documentation

At the moment this documentation is not complete. If you need more help please feel free to contact me with your questions/comments at the following email address: `d00mas@efd.lth.se`