Departamento de Engenharia Informática

Relatório do Projeto de Compiladores 2013/2014
Linguagem imperative Java

Hélio Bento Renato Pires nº 2008117846

João Miguel Rodrigues Jesus nº 2008111667

Índice

Introdução

No âmbito da cadeira de compiladores do curso de Engenharia Informática nos foi proposto desenvolver um compilador para a linguagem Ijava (imperative java) . A linguagem Ijava está contida em java ou seja qualquer programa aceite no ijava tambem funciona em java . O ijava apresenta restrições,pois o programa só pode conter uma classe,so existem arrays 1 dimensão e as funções e os atributos têm que ser públicos . Este projecto foi dividido em 3 metas , portanto o relatório vai seguir esta estrutura, explicando o trabalho desenvolvido em cada uma das etapas como as decisões tomadas. Na Primeira meta (Análise lexical) , utlizámos o lex para identicar os tokens da linguagem . Segunda meta (Análise Sintática e Análise Semântica), na Análise Sintática procedemos à construção de uma gramatica no yacc apartir de uma gramática fornecida na notação EBNF , na Análise Semântica. Na Terceira meta (Geração de codigo final), temos de passar os elementos da Árvore de Sintaxe Abstracta, para código llvm conforme o que este representa, para que depois através de um compilador possamos compilar llvm para código fonte.

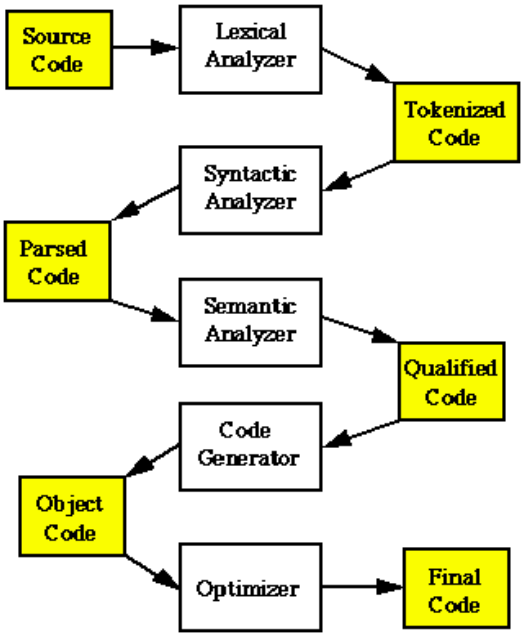

Fig1 : Estrutura típica de um compilador (diversas fases).

Análise lexical e Tratamento de Erros Lexicais

Nesta fase inicial do nosso compilador, são produzidos os tokens da nossa linguagem . Para isso criámos um analisador lexical (usando o lex), capaz de identificar os tokens da linguagem e enviar uma mensagem de acordo com qual fosse identificado para posterior processamento . Quando o nosso analisador lexical encontra um comentário ignora essa linha, caso seja encontrado o "//" e caso seja "*/" fica num estado em que ignora tudo ate encontrar o fim do comentario "/*", caso não seja encontrado gera um erro lexical . Uma mensagem de erro tambem é imprimida se encontrar padrões que não se encaixam como token da linguagem, apresentado a localização do erro .

Tokens da linguagem

A seguir apresentamos os tokens da linguagem, a maioria já está definida no enunciado do projecto e outros tiveram que ser adicionados ou decompostos .

ID : Sequências alfanuméricas que necessariamente começam por uma letra , é definido pela expressão regular : [a-zA-Z_$]([a-zA-Z_$0-9])* .

INTLIT: Sequências de dígitos decimais, e sequências de dígitos hexadecimais (incluindo a-f e A-F) precedidas de "0x". exp regular : (([0-9])+|("0x"[0-9a-fA-F]+)) .
BOOLLIT = "true" | "false"
INT = "int"
BOOL = "boolean"
NEW = "new"
IF = "if"
ELSE = "else"
WHILE = "while"
PRINT = "System.out.println"
PARSEINT = "Integer.parseInt"
CLASS = "class"
PUBLIC = "public"
STATIC = "static"
VOID = "void"
STRING = "String"

DOTLENGTH = “.length”
RETURN = “return”
OCURV = “(”
CCURV = “)”
OBRACE = “{”
CBRACE = “}”
OSQUARE = “[”
CSQUARE = “]”
OP1 = “&&” | “||”
OP2 = “<” | “>” | “==” | “!=” | “<=” | “>=”
OP3 = “+” | “-”
OP4 = “*” | “/” | “%”
NOT = “!”
ASSIGN = “=”
SEMIC = “;”
COMMA = “,”
RESERVED = ++ ; -- ; abstract ; continue ; for ; switch ; assert ; default ; goto ; package ; synchronized ; do ; private ; this ; break ; double ; implements ; protected ; throw ; byte ; import ; throws ; case ; enum ; instanceof ; transient ; catch ; extends ; short ; try ; char ; final ; interface ; finally ; long ; strictfp ; volatile ; const ; float ; native ; super ; null .

O grupo Reserved existe devido à necessidade da linguagem ijava respeitar as regras do java , portanto esses tokens que são palavras reservadas no java não podem ser utilizados na linguagem ijava como um identificador, por exemplo .

Nota : O **OP1** e **Op2** foram divididos em mais operadores para poder respeitar regras de precedência .

Análise Sintática

Nesta fase, em que é preciso resolver questões Sintáticas ou seja como é que é permitido combinar os diversos tokens da linguagem, é preciso uma gramática não ambígua. Para a Análise Sintática do nosso compilador, foi nos fornecido uma gramática, porem esta era ambigua e em notação ENBF,ou seja, não é aplicavel à ferramenta (yacc). Apartir desta, criámos uma gramática válida para o yaac e resolvendo questões como criar condições para ser possivel repetições de expressões utilizando para isso a recursividade a direita. Para lidar com a possibilidade de tokens serem opcionais em determinadas situações foram criados estados adicionais para regras que continham tokens opcionais .

Gramática Fornecida

Start → Program
Program → CLASS ID OBRACE { FieldDecl | MethodDecl } CBRACE
FieldDecl → STATIC VarDecl
MethodDecl → PUBLIC STATIC (Type | VOID) ID OCURV
[FormalParams] CCURV OBRACE { VarDecl } { Statement } CBRACE
FormalParams → Type ID { COMMA Type ID }
FormalParams → STRING OSQUARE CSQUARE ID
VarDecl → Type ID { COMMA ID } SEMIC
Type → (INT | BOOL) [OSQUARE CSQUARE]
Statement → OBRACE { Statement } CBRACE
Statement → IF OCURV Expr CCURV Statement [ELSE Statement]
Statement → WHILE OCURV Expr CCURV Statement
Statement → PRINT OCURV Expr CCURV SEMIC
Statement → ID [OSQUARE Expr CSQUARE] ASSIGN Expr SEMIC
Statement → RETURN [Expr] SEMIC
Expr → Expr (OP1 | OP2 | OP3 | OP4) Expr
Expr → Expr OSQUARE Expr CSQUARE
Expr → ID | INTLIT | BOOLLIT
Expr → NEW (INT | BOOL) OSQUARE Expr CSQUARE
Expr → OCURV Expr CCURV
Expr → Expr DOTLENGTH | (OP3 | NOT) Expr
Expr → PARSEINT OCURV ID OSQUARE Expr CSQUARE CCURV
Expr → ID OCURV [Args] CCURV
Args → Expr { COMMA Expr }

Árvore de Sintaxe Abstracta

A Árvore de Sintaxe Abstracta é onde guardamos a estrutura do nosso código . Na nossa implementação foi possivel utilizar uma única estrutura devido as características da linguagem Ijava . O facto de usarmos apenas uma estrutura facilitou-nos em termos das operações realizadas . Assim, conseguimos através de uma estrutura sintetizar todos os tipos de nós necessários e por consequência as,todas as suas funções da forma mais objectiva possível, sem termos por isso, estruturas redundantes.
Sendo que o yacc, começa por analisar os nós terminais, até juntar - los em operações, expressões, etc…; podemos afirmar que o yacc é um analisador vertical ascendente.

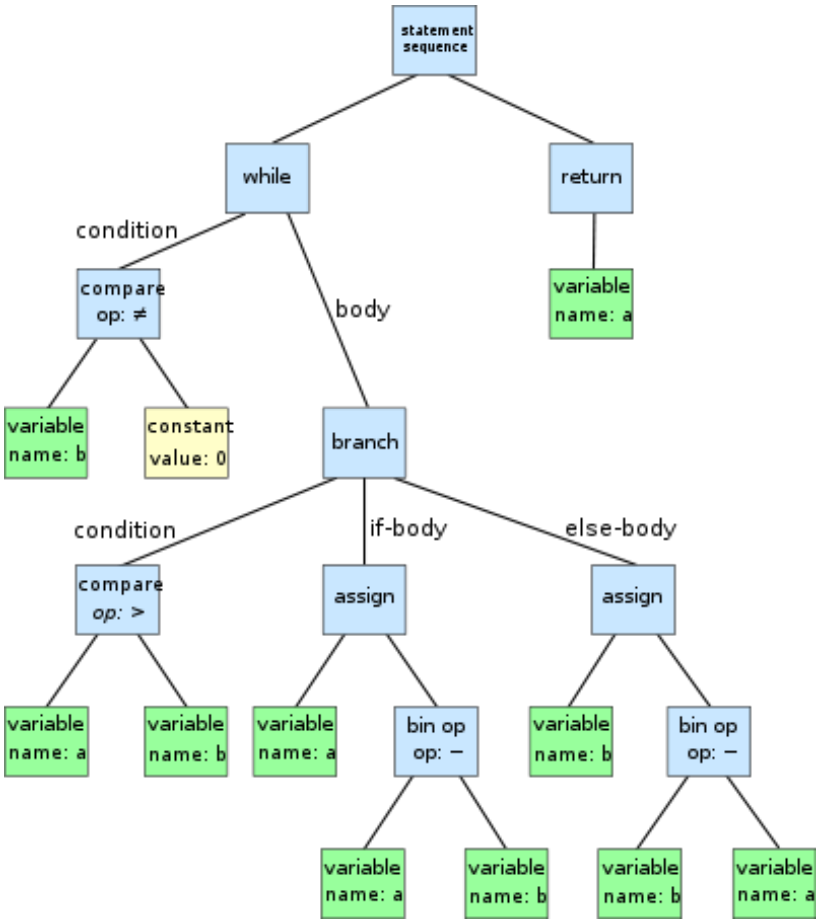

Fig2 : exemplo de arvore de sintax abstracta
fonte: http://en.wikipedia.org/wiki/Abstract_syntax_tree

Análise Semântica

Nas fases anteriores garantimos que o código escrito estava em Ijava .Nesta fase resolvemos questões compatibilidade de tipos , declaração e utlização de variáveis e funções . O que pretendemos nesta fase é garantir que as operações “fazem sentido” ou seja tenham algum significado válido. A maior preocupação nesta fase é identificar os chamados erros semânticos . Ex: atribuir um inteiro a um boolean , usar o mesmo id para variaveis diferentes, etc..

Para poder proceder às verificações necessárias que asseguram a coerência semântica é necessario criar estruturas para guardar essas informações para poderem ser consultadas posteriormente , para isso foi criado as tabelas de simbolos .

Tabela de simbolos

Criámos a tabela de simbolos apartir da Árvore de Sintaxe Abstracta . O processo de criação passa por percorrer a Árvore de Sintaxe Abstracta e criar uma entrada na tabela para cada método ou atributo declarado. A tabela de símbolos neste caso tem que conter todos os métodos e variáveis declarados visto serem todos públicos. Ao termos a tabela de simbolos fica fácil verificar situações em que se invoca métodos não definidos ou variáveis não declaradas .

Geração de código

Nesta última fase do nosso projecto tivemos que usar o conjunto de ferramentas LLVM (low level virtual machine), que apesar de tudo é composto por um compilador C/C++, que a partir de uma linguagem llvm, conseguimos obter uma executável através deste compilador especifico. Assim sendo, o objectivo desta meta, consistia em tentar transformar a Árvore de Sintaxe Abstracta em liguagem LLVM, para que depois possamos utilizar este compilador para transformar num programa executável. Assim sendo e apesar de termos bastantes dificuldades nesta meta conseguimos de certa maneira incompleta, fazer esta transformação em certos casos.

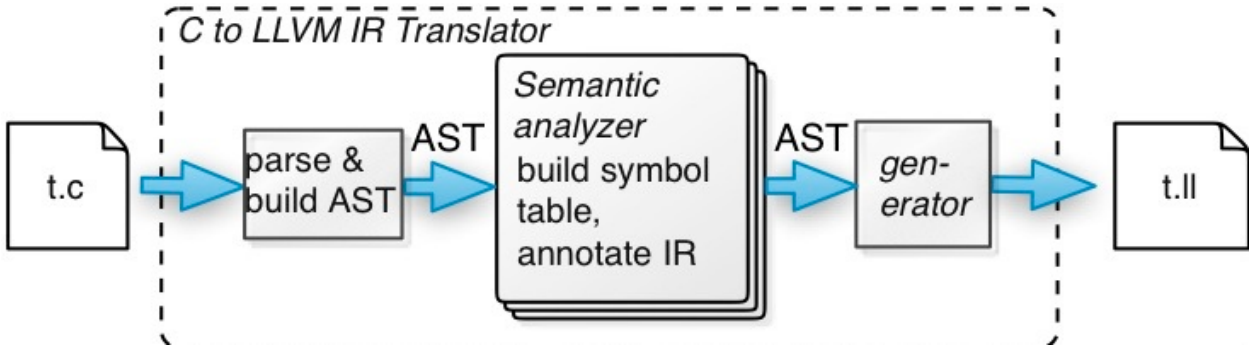

Fig. 3 - Fases Semelhantes ao iJava(Source a LLVM)

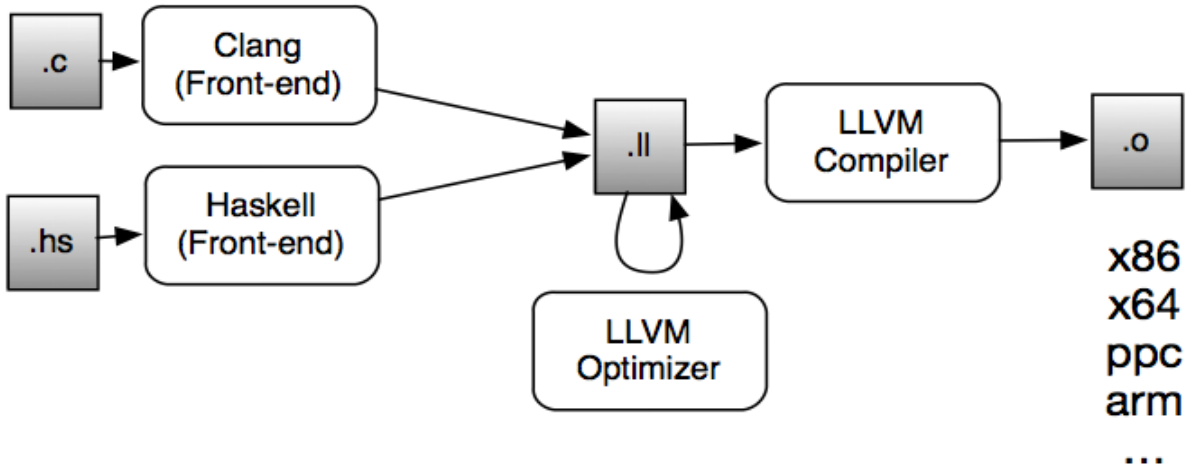

Fig. 4 - Arquitectura (Source a LLVM)

Complicações

Embora tivessemos dificuldades em todas as metas, tivemos alguns problemas na entrega da meta 2, na qual apenas conseguimos ter a Árvore de Sintaxe Abstracta correcta e a Análise Sintáxica, assim com alguns pontos na Tabela de Símbolos. Apenas conseguimos submeter esta meta na totalidade, no Pós Meta, no qual obtivemos os pontos máximos. Já na meta 3, tivemos muitas dificuldades, devido ao facto de termos de aprender a liguagem llvm e de não termos muito tempo para resolver, daí termos apenas 305 pontos de 400 possíveis.Nesta última meta não conseguimos implementar a tempo a criação, modificação de arrays assim como a parte das funções que recebem e retornam arrays, a partir da AST para LLVM.

Conclusão

A realização deste projecto permitiu-nos conhecer melhor uma das ferramentas mais utilizadas durante todo o curso (compilador) . Embora houve dificuldades, em todas as metas, mas foi gratificante pois esse conhecimento nos permite, sermos melhores programadores. Pois o conhecimento dos mecanismos internos dum compilador ajuda - nos a ser mais eficientes quando programamos porque saberemos como estruturar o código de forma óptima para o compilador. Será de grande valor principalmente no debug de aplicações . Acreditamos que o conhecimento adquirido possa ser aplicado tambem não s para desenvolvimento dum compilador em si como tambem em processadores de linguagens, e aproveitar assim das capacidades dos compiladores existentes para obter uma melhor performance .