Compiladores

Relatório do projecto

Bruno José Borges Madureira 2011161942

João Miguel Coimbra Barros da Silva
2011148814

Compiladores, 2 ºSemestre

2013/2014

Indice

Introdução

Neste projecto foi-nos pedido para desenvolver um compilador para uma sub-especificação imperativa da linguagem java, denominada iJava. O desenvolvimento deste foi feito em 3 fases:

- Fase 1: Analisador Lexical .
- Fase 2 Analisador Sintáctico e Semantica:
  - Tratamento da gramática inicial em notação EBNF.
  - Árvore de sintaxe abstracta
  - Tabela de símbolos
  - Tratamento de erros semânticos
- Fase 3: Compilador e Relatório.

A linguagem iJava, como já foi referido anteriormente, é uma linguagem baseada no java mas com um paradigma puramente imperativo rejeitando as funcionalidades de programação orientada ao objecto do Java. Em termos funcionais, esta linguagem é bastante limitada. Apenas suporta tipos inteiros (32 bits) e booleanos permitindo realizar operações aritméticas , passagem de parâmetros pela função main e impressão no ecrã. Outra limitação da linguagem é a de apenas permitir uma classe e esta tem de ter lá dentro obrigatoriamente o metodo main(), podendo ter mais funções.

Para a elaboração deste compilador usamos 2 ferramentas: o “Lex” e o “Yaac”. O “Lex” é um programa que gera analisadores léxicos e o “Yaac” um gerador de analisadores sintácticos.

Este relatório vai ser dividido em 3 secções principais que explicam como procedemos na elaboração de cada meta e para finalizar temos a secção de notas sobre o trabalho feito , bem como uma conclusão e a bibliografia.

O nosso login no mooshack é jcbsilva e as pontuações em cada teste são:

- Meta1: 200/200
- Meta2: 1100/1100
- Meta 3: 400/400

Meta1

Nesta fase começou-se por implementar a detecção de tokens e reconhecimento de comentários.

Para detectar os tokens foi necessário criar um ficheiro lex (.l) contendo para cada token a string correspondente a detectar.

Para a detecção de comentários dividimo-los em 2 tipos, os "//" e os "/**/".

Para implementar a detecção de erros, todos os tokens que não foram aceites pelas regras implementadas, são dadas como erro. Para tratar os comentários "//" simplesmente criamos uma regra lexical que diz que sempre que se encontrar um linha começada por // ignora-se tudo para frente até o /n ser encontrado.

Para comentários com "/* */" criou-se outro estado além do estado default, o estado "COMMENTS" ao qual o lex entra sempre que encontrar um " /*". Dentro do estado "COMMENTS" tudo o que aparecer é ignorado até ser encontrado um "*/ ", quando "*/" for encontrado o lex volta ao estado default, metendo a variável commentCollumn (que é a variável que diz em que coluna foi encontrado o comentário actualmente a ser processado) a 0.

Poderíamos ter criado 2 estados adicionais para fazer a detecção dos comentários, mas achamos muito mais simples tratar os comentários "//" da forma que fizemos.

Para detectar o erro do não fecho do "/*" recorreu-se ao YYSTATE e ao ywrap, que devolve o estado actual e diz se o input terminou, respectivamente. Usou-se uma variável <commentColumn> para receber o número de colunas do começo do último "/*" que o programa leu, para que se não houver um fecho por "*/" ter-se a coluna do" /*" que deu origem ao erro, e uma variável que conta o número de colunas que sucede a um "/*". Se não ocorrer um */ até ao fim do ficheiro então o número da linha de ocorrência do último /* é igual yylineno- commentLine; por sua vez, a coluna é igual a a commentLine. Se ocorrer */, então faz-se reset ao commentLine e ao commentColumn.

O que mudamos nesta meta:

Posteriormente depois das alterações efectuadas na meta 2 vimo-nos forçados a alterar a forma como detectávamos os erros do não fecho do "/*", como só é possível ter um "main" tivemos de chamar o "YY_USER_ACTION" sempre que detectavámos um token "EOF" tanto no estado default como no estado "COMMENTS".

No caso de o programa ser vazio dá erro na última posição do mesmo.

Meta2

Esta meta foi subdividida em 4 fases : Analise Sintáctica, Arvore Sintáctica, tabela de símbolos e tratamento de erros semânticos.

Para a elaboração desta meta seguimos a esta divisão por fases por segmentar o trabalho e ser a divisão mais lógica.

A nossa solução foi altamente inspirada pelo livro "Compilers -Principes, Tecniques and Tools"

Tratamento da gramática inicial em notação EBNF

Na primeira fase "Analisador Sintático" foi-nos dada uma gramática ambígua na notação EBNF. Tivemos problemas de ambiguidade e a necessidade de criar produções extra para os casos que existissem produções opcionais ou múltiplas.

Tivemos de definir as seguinte precedências para retirar a ambiguidade da linguagem e para ficar sintaticamente correcto:

%right ASSIGN
%left OP1OR
%left OP1AND
%left OP2EQ OP2NEQ
%left OP2GEQ OP2LEQ OP2GT OP2LT
%left OP3ADD OP3SUB

%left OP4MUL OP4DIV OP4MOD
%right UNARY
%right NOT
%left DOTLENGTH
%left '['
%nonassoc NPRECEND
%nonassoc ELSE
IF '(' Expr ')' Statement %prec NPRECEND

A associatividade à direita do operador '=' garante que se fosse necessário uma atribuição do tipo "a = b = c" Era feito "a = (b = c);"Tomando 'a' o valor de b, que por sua vez toma o valor de c. É natural que o operador '=' tenha prioridade inferior a todos os outros operadores.

O operador '[' tem de ter associatividade à direita: suponhamos que tínhamos uma matriz e queríamos obter o elemento da posição (2,5). Para tal escreveríamos array[2][5] e o modo de funcionar seria este: primeiro obtemos a linha 2 do array, e dessa linha 2 obtemos o elemento da posição 5 se houvesse associatividade à direita, obteríamos a coluna 5 de 2, ou seja não tinha qualquer sentido lógico.

Já no If-Else, ocorre conflitos "shift-reduce," devido ao facto de o "yacc" não saber se deve proceder:

IF expr IF expr stmt . ELSE stmt -- shift
IF expr IF expr stmt ELSE . stmt -- shift
IF expr IF expr stmt ELSE stmt . -- shift
IF expr stmt . -- reduce

ou dever proceder:

IF expr IF expr stmt . ELSE stmt -- shift
IF expr stmt . ELSE stmt -- reduce
IF expr stmt ELSE . stmt -- shift
IF expr stmt ELSE stmt . -- shift

Para o caso da dupla indexação, procedeu-se à criação de 2 tipos de expressões "Expr" e "Expr1"

Na "Expr" pusemos todas as expressões.

O programa detecta apenas um erro sintático (dando a linha e a coluna desse erro) e termina. Para continuar a detectar erros, era necessário ter no yacc regras de recuperação de erro.

Arvore Sintáctica

Nesta fase era necessário fazer a árvore sintática. De acordo com o enunciado a sintaxe e a semântica são tratadas separadamente, a árvore é construída sem ter em conta a semântica, depois da árvore estar construída é possível percorrê-la e analisar a sua validade semântica, se não for válido é reportado o erro ,com isto ganha-se modalidade e independência entre as fases e não é mais necessário percorrer o código. O único problema desta abordagem é não conseguirmos ter referencia da linha em que foi dado o erro semântico, mas como isso não é pedido este problema não se aplica, no entanto se fosse um compilador para uso "sério" e não apenas um exercício académico este problema era pertinente. Tomámos a opção de fazer uma estrutura para cada produção da gramática.

Tirámos recurso a Union's para construir statements, Expr e Expr1. Usamos Enums para saber qual o tipo de struct ir buscar.

Para fazer declarações de listas de declarações, lista de variáveis globias, locais e de parâmetros, multStatments e multExpressions, usámos listas ligadas.

Tendo em conta o facto de o yacc ser um parser ascendente, tivemos em conta que a produção vazia deve ser a primeira a ser avaliada; portanto, as produções vazias inicializam as listas ligadas com um nó (header) que, aquando da iteração dessa mesma lista, esse nó é saltado.

Observações:

Para guardar booleanos e inteiros, foi usado um char e um char*, respectivamente. O inteiro do IntValue é calculado aquando da construção da tabela de símbolos, sendo que o inteiro, seja decimal, hexadecimal ou octal é convertido no respectivo inteiro. Esta alteração foi efectuada aquando a elaboração da 3ª meta do projecto.

Para as 'Expr' e 'Expr1' definimos ainda um Enum adicional para identificar o tipo dessa mesma entidade.

Para imprimirmos a árvore sintática, recorremos a uma variável global que diz quantos espaços é para imprimir; antes de imprimirmos filhos, incrementámos essa variável e quando acabámos de imprimir os filhos, decrementámos o quanto incrementámos.

No caso dos compound Statments vazios dos If-Else e While, usou-se uma função que simplifica compound Statments, ou seja mete o tipo desses statements a MULT_NULL, o que significa que na impressão da árvore sintática ele vai imprimir Nulo so uma vez.

No caso de compound statements que não esteja no If-Else ou While, como o compound Statement é do tipo IS_MULT_NULL e o next é nulo, não imprime nada.

Tabela de símbolos

Nesta fase era-nos pedido que construíssemos uma tabela de símbolos, para o fazer usou-se a árvore construída na fase 2 e percorreu-se fazendo o seguinte:

Criaram-se 3 arrays estáticos para guardar a informação relativa a variáveis e métodos. O array “fieldTab” guarda as variáveis globais, enquanto o array “methodTab” guarda informação relativa aos métodos. O array ”symP” guarda tudo o que os outros dois arrays guardam; este array symP serve para imprimir a tabela de símbolos de forma coerente, respeitando a ordem de como o código Ijava foi escrito pelo programador. Os outros 2 arrays foram uma forma fácil de não estar sempre a perguntar se as variáveis são locais ou parâmetro, ou globais.

ERROS SEMÂNTICOS

Para esta fase, tirámos partido da árvore sintática e fomos a cada multStatementBody de cada método e verificámos a correctude do programa. É verificada a existência da variável nas variáveis parâmetro do método, de seguida nas variáveis locais do método e só depois na tabela de variáveis globais. O scope da variável foi acrescentado na Meta 3, devido à necessidade de, nessa meta, distinguir as variáveis globais.Tal como dito anteriormente, estes erros não vão conter informação sobre a linha em que estão, já que percorremos a árvore e não o código, como a nossa arvore não faz menção da linha , é impossível dizer a linha em que está o erro.

Meta3

A meta 3 é responsável pela geração do código, caso não sejam passados parâmetros o programa gera o código llvm correspondente. Para gerar este código mais uma vez graças a arquitectura do nosso programa apenas precisamos de percorrer a árvore sintáctica e a tabela de símbolos , não precisando do código original para absolutamente nada. Recorremos também a 2 funções do C, o atoi() e o printf.

Fizemos estruturas, para boolean[] e int[]. Para fazer malloc de um boolean[], alocámos um i8* de n/size_do_Array + 1bytes e depois convertemos em i1*.

Todos os métodos têm valores de retorno por defeito. No caso de estarem declarados como int[] ou boolean[], é devolvido um ponteiro para a struct respectiva com length de zero. Caso sejam inteiros ou booleanos é devolvido uma variável a zero.

Os argumentos do main começam no segundo elemento e não no primeiro, ignorando assim o nome do programa do %.argv.

Para fazermos os operadores '||' e '&&' geramos código do tipo If-Else. No caso do '&&', se o 1º operando for 'true' então o resultado da operação é igual ao valor do 2º operando; senão o resultado é logo 'false' sem necessidade de avaliar o 2º operando. Fez-se semelhante para o '||' mas com 'false' para avaliar o 2º operando e 'true' caso o 1º operando seja 'true'.

Cada método tem pelo menos um statement 'return'.

Para meter correctamente labels para o caso dos If-Else e While e seus pontos de escape, e para numerar correctamente variáveis automáticas, criaram-se 3 variáveis globais para o efeito, que são incrementadas antes do uso das mesmas.

Para gerar o método main, abriu-se uma excepção: no caso de haver um 'return 0' no main, é gerado um 'ret void'. Em qualquer caso é sempre gerado um 'ret void' mesmo que não apareça nenhum 'return 0' pelo meio através de If-Else ou While.

Notas

Para guardar booleanos e inteiros, foi usado um char e um char*, respectivamente. O inteiro do IntValue é calculado aquando da feitura da tabela de símbolos, sendo que o inteiro, seja decimal, hexa ou octal é convertido no respectivo inteiro. Esta alteração foi efectuada aquando da feitura da 3ª meta do projecto

Para as 'Expr' e 'Expr1' definimos ainda um Enum adicional para identificar o tipo dessa mesma entidade

É de notar que, chamarmos uma função que retorna void, o erro dado não é o esperado.

Não é erro semântico o facto de não haver método main.

Metodos e variaveis globais não podem ter o mesmo nome.

Conclusão

Este trabalho deu-nos mais noção sobre o funcionamento interno de um compilador, ensinou-nos como funcionam geradores de analisadores lexicais e sintácticos e que estruturas de dados um compilador cria durante o seu funcionamento.

Bibliografia

Levine, Jhon . Flex & Bison.

T.Niemann. A Compact Guide to Lex & Yacc.

Apell, Andrew W. Modern Compiler Implementation in Java, Second Edition.

(Varios autores) Modern Compiler Design, Second Edition.

(Varios autores) Compilers -Principes, Tecniques and Tools, Second Edition

. A. Appel, Modern compiler implementation in C. Cambridge Press, 1998.