· U  C ·

Universidade de Coimbra

Compiladores

Compilador iJava

Grupo Google

Autores:
João L. Cardoso [i]
Paulo Pereira [ii]

2 de Junho de 2014

[i] nº2011151968
[ii] nº2011164879

Índice

1 Introdução

Este projeto prático consiste no desenvolvimento de um compilador para a linguagem iJava. Neste compilador os ficheiros aceites apenas poderão possuir uma classe, tendo obrigatóriamente um método *main* e podendo ter outros métodos e atributos, todos eles estáticos.

A linguagem iJava utilizada é semelhante à linguagem java contendo algumas alterações. Nesta linguagem é possível utilizar variáveis e literais do tipo inteiro de 32 bits com ou sem sinal e booleano. Podemos também utizar array de valores do tipo inteiro e booleano apenas de uma dimensão. Além destes tipos de arrays também é possivel usar o array de String, no entanto este tipo não pode ser criado apenas pode ser recebido por parâmetro no método main e para manipular o array de String recebido exite o método *Integer.parseInt()* que converte um elemento deste array para um literal inteiro.

A linguagem implementa expressões aritméticas e lógicas e operações relacionais simples bem como operações de controlo (if-else e while). Os métodos poderão retornar valores do tipo inteiro ou booleano.

O compilador desenvolvido neste trabalho foi implementado em C com recurso as ferramenta *lex*, *yacc* e *LLVM*.

Este projeto será dividido nas seguintes fases:

- Analisador lexical (verifica se os tokens introduzidos são válidos)
- Analisador gramatical (verifica se os tokens seguem a gramática definida)
- Construção da árvore de sintaxe abstrata e da tabela de símbolos
- Geração de código (cria um ficheiro em código *LLVM* que implementa as mesmas funcionalidades do programa de entrada)

2 Analisador Lexical

O analisador lexidar implementado neste compilador foi construido com recurso à ferramenta *lex*. Os tokens válidos neste compilador são apresentado em seguida.

- ID (sequência de alfanúmerica começada por uma letra, onde os símbolos "_"e "$"contam como uma letra)
- INT (*inteiro em escala decimal ou hexadecimal*)
- BOOL (*boolean*)
- VOID (*void*)
- STRING (*String*)
- INTLIT (sequência de dígitos decimas, hexademais e octais)
- BOOLLIT (*true* e *false*)
- NEW (*new*)
- CLASS (*class*)
- PUBLIC (*public*)
- STATIC (*static*)
- IF (*if*)
- ELSE (*else*)
- WHILE (*while*)
- PRINT (*System.out.println*)
- PARSEINT (*Integer.parseInt*)
- DOTLENGTH (*.length*)
- RETURN (*return*)
- OCURV (*(*)
- CCURV (*)*)
- OBRACE ({)
- CBRACE (})
- OSQUARE ([)
- CSQUARE (])

- NOT (*!*)
- ASSIGN (=)
- SEMIC (*;*)
- COMMA (,)
- OP11 (||)
- OP12 (*&&*)
- OP21 (<, >, <= e >=)
- OP22 (== e *!*=)
- OP3 (+ e -)
- OP4 (*, / e %)
- RESERVED (os restantes tokens utilizados em java e não referidos acima não serão aceites no compilador deste projeto, contudo serão identificados por este token)

Os literais inteiros podem ser uma sequência de dígitos decimais, hexadecimais e octais. Os literais inteiros são hexadecimais se começarem pela sequência *0x* e a seguir tiverem uma sequência de numeros ou das seguintes letras [A-F], maisculas ou minusculas. Os literais octais são identificados por terem mais que um elemento e começarem com o elemento *0* e a seguir tiverem um número contido na seguinte sequência 0-7. Os seguintes tokens (*op11*, *op12* e *op21*, *op22*) resultam da divisão dos tokens (*op1* e *op2*). Esta mudança nos tokens resultou no agrupamento de símbolos com a mesma precedência e que podem ser aplicados ao mesmo tipo de dados. Este agrupamento permite à análise sintática detetar facilmente a precedência de operadores - operadores de multiplicação e divisão têm precedência sobre operadores de adição e subtração, por exemplo.

O compilador tem dois estados: o estado inicial, para detecção de tokens de código iJava, e um estado destinado à detecção de comentários multi-linha. O compilador entra neste último estado se detetar o início de um comentário (*/**), e ignora todo o input até encontrar o final do comentário (**/*), momento em que volta ao estado inicial.

Caracteres brancos e mudanças de linha são ignorados. Qualquer outro caracter detectado resulta num erro de caracter inválido.

Para o caso de o intrepertador detetar um caracter inválido, é feita uma contagem de linhas e de colunas para o mostrar não só o caracter inválido mas também a linha e a coluna onde se encontra o erro. Também é mantido em memória a linha e coluna do inicio do token actual. Isto permite no caso de uma expressão não passar a análise lexical mostrar não só o tipo de erro mas também a linha e a coluna onde ocorre.

3 Analisador Gramatical

O analisador semântico foi implementado através do interpretador lexical descrito anteriormente e com recurso à ferramenta *yacc*. A seguinte grámatica define a sintaxe utilizada para o desenvolvimento deste compilador da linguagem iJava.

```
    program → CLASS ID OBRACCE declarations CBRACE
    declarations → declarationList | λ
    declarationList → declaration declarationList | declaration
    declaration → fieldDecl | methodDecl
    fieldDecl → STATIC varDecl
    methodDecl → PUBLIC STATIC type ID OCURV params CCURV OBRACE statements
CBRACE
    statements → varList stateList | varList | stateList | λ
    params → STRING OSQUARE CSQUARE ID | paramList | λ
    paramList → param COMMA paramList | param
    varList → varDecl varList | varDecl
    param → varType ID
    varDecl → varType ids SEMIC
    ids → ID COMMA ids | ID
    stateList → statement stateList | statement
    statement → ifState ELSE statement | ifState | WHILE OCURV expr
CCURV statement | OBRACE stateList CBRACE | OBRACE CBRACE | PRINT OCURV
expr CCURV SEMIC | ID ASSIGN expr SEMIC | ID OSQUARE expr CSQUARE ASSIGN
expr SEMIC | RETURN optionalExp SEMIC
    ifState → IF OCURV expr CCURV statement
    optionalExp → expr | λ
    expr → exprindex | exprnoindex
    exprnoindex → NEW numType OSQUARE expr CSQUARE | expr OP11 expr
| expr OP12 expr | expr OP21 expr | expr OP22 expr | expr OP3 expr
| expr OP4 expr | OP3 expr | NOT expr
    exprindex → exprindex OSQUARE expr CSQUARE | ID | INTLIT | BOOLLIT
| expr DOTLENGTH | OCURV expr CCURV | PARSEINT OCURV ID OSQUARE expr
CSQUARE CCURV | ID OCURV optionalArgs CCURV
    optionalArgs → args | λ
    args → expr COMMA args | expr
    type → VOID | varType
    varType → numType | numType OSQUARE CSQUARE
    numType → INT | BOOL
```

A seguinte tabela define a ordem de precedência bem como a ordem de associação, isto é se associa à direita ou à esquerda, utilizada neste trabalho.

Operador	**Associatividade**
.legth, [	esquerda
!, + (sinal positivo), - (sinal negativo)	direita
new, (	direita
*, /, %	esquerda
+, -	esquerda
<, >, <=, >=	esquerda
==, !=	esquerda
&&	esquerda
||	esquerda
=	direita

Para resolver o problema de expressões de criação de arrays multi-dimensionais, como por exemplo "new int[4][2]", na regra das expressões (*expr*) separamos as expressões em dois conjuntos: as expressões que podem ser indexadas (*exprindex*) e as expressões que não podem ser indexadas (*exprnoindex*). É de ter em atenção que a gramática tal como está aceita expressões de indexação de arras multi-dimensiomais, como por exemplo "a[2][1]", contudo este tipo de erro deve ser detentado pelo analisador semântico e não pelo gramática.

Originalmente as expressões que utilizam os tokens *OP11*, *OP12*, *OP21*, *OP22*, *OP3*, *OP4* que estão na regra *exprnoindex* estavam numa regra à parte que só tinha expressões com operadores, contudo tivemos de alterar porque como estava não estava a detetar a precedência de cada sinal, que está definida na tabela acima. Por isso a solução foi passar as expressões com operadores para as regras das expressões.

É de referir que na gramática acima não estão presentes os nonassocs utilizados. Foram adicionados dois *nonassocs* para dar mais prioridade ao statement que do if-else do que ao if, isto porque caso contrário o analisador ao encontrar um if-else não saberia a que statement *if* pertencia se era ao if-else ou ao if. Além destes dois *nonassocs* adicionamos outro nas expressões para dar a mesma prioridade do not às expressões com sinal positivo ou negativo, isto para a prioridade do not e dos sinais ficarem de acordo com o que está definido na liguagem Java.

Como já foi referido na secção anterior nesta parte do projeto tivemos que alterar o ficheiro *lex* para permitir no caso de erros na gramática mostrar o tipo de erro e também a linha e a coluna do início do token que deu erro. No caso de erros o analisador pará no primeiro erro.

4 Árvore de Sintaxe Abstracta

A nossa árvore de sintaxe abstracta é consituída essencialmente por listas ligadas de estruturas hierarquizadas.

A árvore é representada pela estrutura *Program*. O *Program* contem informações relativas à classe - nome e uma lista ligada de declarações. É a prórpria estrutura *Declarations* que permite ser ligada para criar essa lista (contem um ponteiro para o elemento seguinte ou *NULL*). Esta propriedade é sempre usada nos tipos de estruturas para criar todas as listas ligadas.

Cada *Declaration* pode ser uma declaração de variável, *VarDecl*, ou declaração de método, *MethodDecl*. Cada *VarDecl* é consituída por um tipo e uma lista ligada de nomes. Cada *MethodDecl* é constituída por um tipo, um nome, duas listas ligadas de *VarDecl* (para representar parâmetros e variáveis separadamente), e uma lista ligada de *Statements*.

Há vários tipos de *Statements*. A cada tipo corresponde uma estrutura diferente. Estas estruturas são essencialmente constituídas por expressões e novos statements (como, por exemplo, para o caso do código a ser executado numa while loop), conferindo assim uma organização recursiva à árvore a partir deste nível da estrutura.

As expressões, *Exp*, seguem o mesmo esquema de *Statements*: podem ter vários tipos, e a cada tipo corresponder uma estrutra diferente. No entanto, neste caso, a larga maioria dos tipos de expressões recorre à mesma estrurura, *Oper*, pois grande parte das expressões podem ser tratadas como um operador genérico e representadas por esta estrutura. A estutura *Oper* pode conter um número indeterminado de expressões dependendo do tipo (contem uma lista ligada), o que confere à estrutura *Exp* também uma organização recursiva.

A impressão da árvore (ativada com o comando opcional *-t*) é feita percorrendo àrvore um única vez através de uma depth-first search. Para cada elemento da árvore, a informação relativa a si é impressa, seguida da informação relativa aos seus descendentes. A variável *identation* é usada para registar o número de espaços que devem ser impressos no inicio de cada nova linha - e assim conferir identação ao output.

É de salientar que os literais inteiros são guardado como um char*, isto porque o valor pode ter sido escrito em escala decimal, hexadecimal ou octal e na impressão da árvore queremos imprimir a representação original.

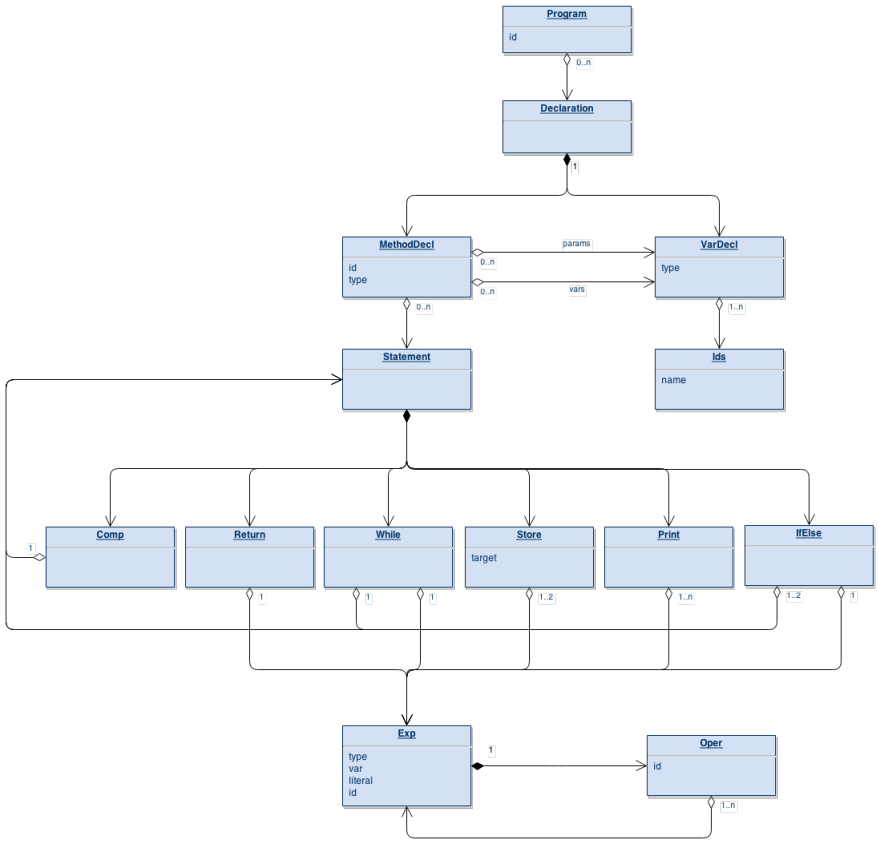

Figura 1: Estrutura resumida da árvore

5 Tabela de Símbolos

A tabela de símbolos é constituida por uma lista ligada de variávies globais e de métodos em que cada método terá uma lista ligada das suas variáveis locais bem como dos seus parâmetros. Em termos de complexidade algorítmica, qualquer pesquisa de uma variável na tabela de símbolos corresponderá assim sempre a uma pesquisa linear. Melhorar a complexidade de pesquisa só seria possível usando estruturas muito mais complicadas de implementar em C, como por exemplo estruturas que façam recurso a *hashing* dos nomes dos síbmbolos ou que permitam pesquisas binárias, pelo que decidimos que esta solução era a mais indicada tendo em conta a dimensão do projeto.

Uma vez que a árvore de sintaxe abstracta preenche as características ditas anteriormente não vimos necessidade de criar uma extrutura à parte para a tabela de símbolos: a árvore de sintaxe abstrata foi criada com recurso a listas ligadas, sendo que os métodos e as variáveis globais pertencem à lista ligada das *Declarations* e dentro de cada método existe duas listas ligadas uma para as variàveis locais e outra para os paramêtros. Por isso, focámos os nossos esforços na criação de funções eficientes de pesquisa de nomes na árvore de sintaxe abstracta. Assim, implementámos uma função para pesquisa de declarações na classe, *findFieldType*, e uma para pesquisa de nomes disponíveis num dado método, *findVarType* (que faz uso da primeira função internamente).

6 Analisador Semântico

A detecção de erros semânticos é feita em duas fases:

- Detecção de símbolos duplicados
- Detecção de utilização de símbolos não existentes ou incompatibilidades de tipos em operações

Estas duas fases são realizadas antes da impressão da tabela de símbolos no ecrã pois caso de existirem erros, a tabela não deverá ser impressa. Para a primeira fase do analisador semântico é feita uma pesquisa linear nas listas ligadas e, para cada símbolo, verifca-se se nenhum dos anteriormemente definidos no mesmo espaço de variáveis apresenta o mesmo nome. Caso haja, é emitido um erro e a pesquisa para. Por isso, caso não haja erros apresenta uma complexidade de $O({}^{n}C_{2})$.

A segunda fase é mais complexa. Para cada statement, verificamos se há concordância do tipo de variável associado às expressões usadas no statement com a operação a ser efectuada pelo mesmo. O tipo de variável de cada expressão depende da sua categoria. Por exemplo, um literal *true* é do tipo booleano, enquanto um operador de adição devolve sempre um inteiro e requere que as duas expressões somadas retornem inteiros também.

Assim que o primeiro erro é detectado, a mensagem de erro associada é emitida e a variável global *hasErrors* é definida para *1*. Esta variável é usada pra impedir que outros erros sejam emitidos, ou que o código seja gerado havendo erros, pois o compilador verifica sempre se a variável não foi definida antes de imprimir texto.

7 Geração de Código

A geração de código resulta na tranformação de input recebido num ouput que implementa as mesmas funcionalidades do programa de entrada (isto se o programa de entrada se apresentar correto) representado na linguagem intermédia *LLVM*. Esta linguagem foi originalmente concebida para C e C++, pelo que apresenta algumas semelhanças com estas duas linguagens.

As funções não podem receber *String arrays* por parâmetro e, que no caso de o tipo de retorno seja *void*, não devem retornar nenhum valor. A excepção a esta regra é a função *main*:

- Recebe uma *String[]* como parâmetro. Em LLVM é representada (tal como em C) através de dois inteiros: o primeiro representa o tamanho da array e o segundo o ponteiro para ela.
- Está definido em Java como sendo do tipo *void*, porém em LLVM tem o tipo *i32* e retorna zero por defeito. Caso contrário, indicaria que ocorreram erros na execução do programa. Usar o tipo *void* não seria possível, pois funções do tipo *void* retornam valores arbitrários.

É de salientar que em C a *String[]* (ou mais corretamente o *char***) recebido possui no primeiro elemento o nome do ficheiro. Para resolver este problema os acessos à array são feitos ao próximo elemento.

Os tipos de dados que estamos a usar para as variavéis são *i32* para inteiros, *i1* para booleanos, *i8** para String, *i32* para arrays de inteiros, *i1** para arrays de booleanos e *i8*** para arrays de String. As váriavéis são inicializadas tal como em Java: no caso de ser uma váriavél global literal a zero e no caso de ser um array a *null*.

A mémoria para guardar o valor das variáveis é alocada no momento da sua declaração. Quando se predente guardar valores nessas váriavéis é feito um store para guardar o valor, sendo que as nossas variáveis são ponteiros. No caso dos arrays, quando é detetado um *new int* ou *new bool* mémoria é alocada com recurso à função *malloc* do C. A função de alocação de mémoria do LLVM não pode ser usada neste caso porque a aloca na stack frame e os dados perder-se-iam quando se sai da função. A função *malloc*, por outro lado, reserva memória na heap e assim está sempre acessível. Para guardar o tamanho destes array no momento da alocação de mémoria são reservados 4 bytes extra e guardado no início do array o tamanho (em *boolean[]* é necessário fazer o bitcast de *i8** para *i32**). Assim para aceder ao tamanho é só necessário aceder à primeira posição e aos elementos somar 4 bytes ao index.

No caso das expressões com o operador *and* e *or* a segunda expressao só pode ser realizada se na primeira não for false e true respectivamente. Para isto acontecer implementados uma condição que só corre a segunda expressão se a primeira não satisfazer a condição. A solução implementada é muito semelhante à soluçao para o statement *if-else*.

Para o retorno de valores foi escolhida uma abordagem de apenas um ponto de saída da função. Para isso, no momento da declaração da função é declarada também uma variável para guardar o valor de retorno. Quando é encontrada uma instrução de retorno no código de entrada, é substituída por uma instrução para guardar na variável o valor a retornar e outra para saltar para a label de retorno. No final da chamada de cada função é sempre feito um salto para a label de retorno, mesmo que não haja nenhuma instrução de retorno no código de entrada (pois é requerido na linguagem do LLVM).

Uma vez que a seguir a uma instrução de salto (br) tem de estar a declaração de uma label, esta abordagem apresentava problemas quando havia mais de uma dessas instruções para a mesma label. No entanto, é garantido que a segundo instrução é inacessível. Por isso, A solução encontrada foi não gerar código inacessível.

Os parâmetros recebidos pelas funções são copiados para uma váriavél. Isto acontece para no caso de uma alteração do valor edste parâmetro, tal como por exemplo $x = x + 1$ em que x é um inteiro recebido por parãmetro este valor não podia ser alterado. Para não causar problemas com o nome das váriavéis declaradas foi adicionado a palavra *.temp* no nome da variável (estas palavras não são aceites pela gramática definida acima), deste modo a funçaõ recebe a variável, aloca memória, tal como na declaração de variáveis, e guarda o valor recebido por parâmetro na nova variável.

É de salientar que todas as váriaveis auxiliares utilizadas na geração de código não são aceites na grámatica definida acima, isto para não causar problemas com as variáveis que vêm no ficheiro de entrada.

8 Pós Output

No final da execução do compilador, a memória alocada para a àrvore de sintaxe abstracta - a única estrutura que criámos - é libertada. Para isso,a àrvore é percorrida um única vez usando depth-first search. Para cada elemento da árvore, é delegado aos seus descendentes que libertem a sua memória e dos seus próprios descendentes. Só depois é libertada a memória relativa ao elemento em questão.

No final de libertar toda a memória reparamos que ainda existia memória que não tinha sido libertada. Depois de alguns testes, nomeadamente de retirar todas as funções que estavam a alocar memória, incluido o *(char*)strdup(yytext)* no ficheiro *lex* observamos que a mesma memória não libertada anteriormente referida estava presente também neste teste. Pelo que concluimos que internamente as ferramentas utilizadas como recuros isto é o *lex* e/ou o *yacc* estão a alocar memória e depois não estão a libertar essa memória.

9 Conclusão

No final, a grande maioria do código acabou por estar implementado na linguagem C, constituindo o código *lex* e *yacc* partes fulcrais mas diminutas em extensão de código. O nosso compilador passou todos os testes publicados no Mooshak na sua totalidade.

A compilação do próprio compilador poderia ser mais rápida tivéssemos colocado as funções C em ficheiros separados, em vez de as incluirmos em headers. No entanto, o compilador é tão simples que não traria vantagens visíveis para o processo de desenvolvimento.

Outro elemento possível de melhorar seria o tratamento de literais de inteiros inválidos ao nível da gramática, ao invés de o fazer ao nível semântico. Seria uma solução mais simples, mas o enunciado pede que seja feita a nível semântico. Finalmente, também traria vantagens ao compilador na gramática definir a obrigatoriedade de existir uma função *main* que seria do tipo *void* e que seria a única função que poderia receber como parâmetros *String arrays*.